TEKNILLINEN KORKEAKOULU
Tietotekniikan osasto

Diplomityö

Pete Hakkarainen

Tosiaikaiset XML-dokumenttien muunnokset verkkopalvelun toteutuksessa

22.6.2004

Valvoja: prof. Eljas Soisalon-Soininen, Teknillinen korkeakoulu

Ohjaaja: fil. kand. Kari Makkonen, FD Finanssidata Oy

TEKNILLINEN KORKEAKOULU
Tietotekniikan osasto

DIPLOMITYÖN TIIVISTELMÄ

Tekijä
Pete Hakkarainen

Päiväys
22.6.2004

Sivumäärä
85

Työn nimi
Tosiaikaiset XML-dokumenttien muunnokset verkkopalvelun toteutuksessa

Professuuri
Ohjelmistojärjestelmät

Koodi
T-106

Työn valvoja
prof. Eljas Soisalon-Soininen

Työn ohjaaja
fil.kand. Kari Makkonen

Portaali on yhtenäinen käyttöliittymä organisaation tarjoamiin palveluihin. Portaalipalvelin hyödyntää muiden järjestelmien tuottamaa tietoa tätä käyttöliittymää muodostaessaan. Tässä työssä tutkitaan kahta erilaista tapaa käsitellä portaalissa muiden järjestelmien tuottamia XML-dokumentteja ja toteuttaa niiden sisältämiä tietoja hyödyntäen XHTML-pohjainen käyttöliittymä. Valituilla tekniikoilla toteutetaan joukko kohdeyrityksen verkkopalvelussa tyypillisiä käyttöliittymärakenteita. Ratkaisuja arvioidaan kehittämisen helppouden ja tehokkuuden sekä toisaalta ajonaikaisen suorituskyvyn kannalta.

Molemmat ratkaisut toteutetaan J2EE-alustalle. Ensimmäisessä ratkaisussa käytettävä XSLT-kieli on suunniteltu yleisiin XML-dokumenttien muunnoksiin. XSLT-muunnoksissa käytetään Apache-projektissa kehitettävää XSLTC-kääntäjää. Vertailukohdaksi toteutetaan sama toiminnallisuus JSP-sivuina ja tässä hyödynnetään erityisesti JSTL-laajennuskirjastoa. Ratkaisuja testataan Tomcat-palvelinohjelmistolla.

Kehittämisen näkökulmasta tekniikat soveltuvat tässä työssä kuvattavien tilanteiden ratkaisemiseen kohtuullisesti. Molemmissa esiintyy ongelmia ja ilmaisuvoima ilman erilaisia laajennuksia ei ole täysin tyydyttävä. Suorituskyvyn ja muistinkäytön suhteen syntyy selvä ero XSLTC-pohjaisen ratkaisun eduksi. Lopuksi analysoidaan tuloksia tarkemmin ja esitetään vaihtoehtoisia ratkaisuja ja mahdollisia syitä JSP-ratkaisun heikolle suorituskyvylle.

Avainsanat: www, portal, xml, xslt, jsp, jstl, java, j2ee, xsltc

HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author
Pete Hakkarainen

Date
22nd June, 2004

Pages
85

Title of thesis
Real-time XML-transformations in Web Application Implementation

Professorship
Software Systems

Professorship code
T-106

Supervisor
prof. Eljas Soisalon-Soininen

Instructor
fil.kand. Kari Makkonen

A portal is a unified user-interface to services provided by an organization. While rendering this user-interface, a portal application takes advantage of information provided by other enterprise information systems. In this thesis I describe two possible solutions for developing an XHTML-based user-interface by manipulating and transforming XML-documents produced by other systems and applications. The chosen technologies are applied to a set of typical structures present in the current web-based services of the reference enterprise. The solutions are analyzed with respect to ease and efficiency of development and run-time efficiency.

Both solutions are built on the J2EE-platform. XSLT-technology, used in the first solution, is designed for general transformations of XML-documents. XSLT-transformations are carried out with help of the XSLTC-compiler developed by the Apache Software Foundation. This solution is compared to one based on JSP-pages and especially JSTL extension tag library. Both solutions are tested with the Tomcat servlet container.

Both technologies proved to be appropriate solutions for the problems presented in this thesis. However, both come with some bugs and annoyances, and the expressive power without using extensions could be better. The XSLTC-based solution was the winner with respect to both speed and memory allocation. Results are more thoroughly analyzed at the end of the thesis. Also alternative solutions are introduced and the inefficiency of the JSP-based solution is discussed.

Keywords: www, portal, xml, xslt, jsp, jstl, java, j2ee, xsltc

Sisällys

Kuvat

Termit ja lyhenteet

API	Application Programming Interface
CGI	Common Gateway Interface; rajapinta, jonka kautta voidaan laajentaa Web-palvelimen toiminnallisuutta. CGI:n vastine J2EE-ympäristössä on Servlet-rajapinta.
DOM	Document Object Model; W3C:n määrittämä ympäristöriippumaton oliomalli rakenteisille dokumenteille.
DSSSL	Document Style Semantics and Specification Language; Scheme-ohjelmointikieleen perustuva tyylimäärittelykieli, jonka käytännössä CSS ja XSL ovat korvanneet.
EJB	Enterprise JavaBean; liiketoimintakomponentti J2EE-arkkitehtuurissa.
HTML	Hyper-Text Markup Language; WWW-sivujen kuvaamiseen käytetty sekä ulkoasua että tiedon rakennetta kuvaava kieli.
InfoSet	XML Information Set; eräs tietomalli XML-dokumenteille, jonka mukainen rakenne syntyy mm. XML Schema –validoinnin yhteydessä.
ISO	International Organization for Standardization
J2EE	Java 2 Enterprise Edition; joukko rajapintoja, jotka yhdessä määrittävät Java-kieleen perustuvan monitasoarkkitehtuurin käyttöliittymän, liiketoimintakomponentit, viestinnän ja järjestelmäpalvelut.
JAXP	Java API for XML Processing; joukko standardoituja rajapintoja XML-dokumenttien käsittelyyn Java-kielellä mm. SAX, DOM ja XSLT –tekniikoilla.
JCP	Java Community Process; yhteisö, jossa standardoidaan Java-tekniikoita.
JSP	JavaServer Pages; käyttöliittymäsivujen toteutustekniikka J2EE-ympäristössä.

JSTL JSP Standard Tag Library; JCP-yhteisössä standardoitu JSP-tagikirjasto, jonka tarkoitus on laajentaa JSP:n yleistä toiminnallisuutta siten, että ylläpidettävyys säilyy hyvänä. ks. JSP

JVM Java Virtual Machine; Java-virtuaalikone, laitteistoriippumatonta Java-tavukoodia suorittava ohjelmisto.

MVC Model-View-Controller; malli käyttöliittymän muodostamisen (view), liiketoimintalogiikan (model) ja käyttäjän antamien käskyjen käsittelyn (controller) erottamiseksi toisistaan.

PSVI Post Schema Validation InfoSet; XML-dokumentin validoinnissa syntyvä InfoSet-tietomallin mukainen kuvaus dokumentin sisällöstä. ks. InfoSet.

SAX Simple API for XML Processing; XML-dokumentin suoraviivaiseen läpikäyntiin ja tapahtumien laukaisemiseen erilaisiin rakenteellisiin elementteihin törmättäessä perustuva jäsennysrajapinta.

Servlet J2EE-rajapinta Web-palvelimen toiminnallisuuden laajentamiseen.

SGML Standard Generalized Markup Language; eräs merkintäkielten metakieli, josta XML on osajoukko.

SOA Service-Oriented Architecture; palvelukeskeinen arkkitehtuuri. Ohjelmisto- tai järjestelmäarkkitehtuuri, jossa komponentit (palvelut) ovat mahdollisimman riippumattomia toisistaan, tehokkaasti uudelleenkäytettäviä ja rajapinnoissa pyritään yksinkertaisuuteen. Tähän liittyy myös palveluhakemisto, josta palveluita voidaan etsiä tietämättä etukäteen niiden todellista sijaintia tai tarjoajaa.

tagi XML-dokumenteissa: elementtien alkamisen ja loppumisen merkitsevä rakenne; <tagi> sisältö </tagi>

JSP-sivulla: Java-kielellä toteutetun *tagikirjaston* yhden komponentin toteutus. Sekaannusten välttämiseksi näitä olisi hyvä kutsua *toiminnoiksi*.

Turing-täydellisyys Järjestelmien ja formaalien kielten laskennalliseen ilmaisuvoimaan liittyvä ominaisuus. Useimmat ohjelmointikielet, samoin kuin tietokoneet ovat yleisesti Turing-täydellisiä. Mitään toteutuksia, jotka ylittäisivät tämän ilmaisuvoiman, ei tunneta.

UCS Universal Multiple-Octet Coded Character Set; ISO/IEC 10646:2000 –standardin määrittämä 31-bittinen merkistö, jonka ensimmäiset 65536 merkkiä ovat Unicode-standardin mukaiset. UCS on XML-dokumenttien merkistö.

Unicode Unicode Consortiumin 16-bittinen "kaikki" maailman kielet kattava merkistöstandardi. 65536 numeron avaruus ei kuitenkaan riitä käytännössä kaikkien kielten kaikille merkeille.

URI	Uniform Resource Identifier; merkkijono, joka identifioi tietyn abstraktin tai fyysisen resurssin. URI:n erikoistapaus URL kuvaa tiettyä verkkoresurssia.
US-ASCII	128 merkkiä kattava merkistö, joka sisältyy esim. laajempiin ISO-Latin-1 ja Unicode –merkistöihin.
UTF-8	UCS Transformation Format for 8-bits; Unicode ja UCS-merkistöjen koodaustapa, jossa yksi merkki ilmaistaan 1-6 oktetilla.
W3C	World-Wide Web Consortium
XHTML	Extensible Hypertext Markup Language; XML-kielen syntaksia noudattava HTML-sivunkuvauskielen versio.
XML	Extensible Markup Language; kieli (metakieli) rakenteisen tiedon kuvaamiseen.
XML Query	Kieli XML-dokumentteihin kohdistuvien kyselyiden määrittelyyn.
XPath	XML Path; kieli XML-dokumentin sisältöön viittaamiseen. XPath on käytössä esim. osana laajempia XSLT ja XQuery-kieliä.
XQuery	ks. XML Query
XSL	Extensible Stylesheet Language; kieliperhe XML-dokumenttien muunnoksiin ja ulkoasun muotoiluun. ks XSLT.
XSLT	XSL Transformations; XSL-kieliperheen osa, jolla voidaan muokata XML-dokumentista toinen XML- tai muu dokumentti.
XSLTC	XSLT Compiler; Apache-projektissa kehitettävä avoimeen lähdekoodiin perustuva XSLT-dokumentit suoraan Java-tavukoodiksi kääntävä ohjelma.

Alkusanat

Tämä diplomityö on tehty tammi-kesäkuussa 2004 FD Finanssidata Oy:lle.

Erityisesti haluan kiittää Kari Makkosta työn ohjaamisesta ja tukemisesta myös projektipäällikön ominaisuudessa sekä valvojana toiminutta Eljas Soisalon-Soinista joustavasta asennoitumisesta ja työn rakenteeseen liittyneistä näkemyksistä. Myös muut työhön liittyneiden projektien projektipäälliköt sekä esimieheni ansaitsevat kiitokset ajan järjestämisestä työn tekemiseen.

Haluan kiittää myös vanhempiani erityisesti kiinnostuksesta työn etenemistä kohtaan sekä ystäviäni, jotka ovat auttaneet minua ajattelemaan muutakin työn ohella.

Ja vielä kiitos Joshua W. Scott:lle englanninkielisen tiivistelmän kieliasun tarkastamisesta.

Helsingissä 23.6.2004 Pete Hakkarainen

Pete Hakkarainen

1. Luku - Johdanto

Tiedonvälitys koki mullistuksen 90-luvun loppupuoliskon aikana uudenlaisten elektronisten ratkaisujen vallatessa alaa perinteisiltä medioilta. Tämän muutoksen takana oli ja on Internet, jonka läpimurto tapahtui pitkälti World Wide Web – tekniikan ansiosta. Tämän diplomityön kohdeyritys oli yksi ensimmäisiä suomalaisia yrityksiä, jotka lähtivät leveällä rintamalla mukaan verkkopalveluiden kehittämiseen.

Verkkopalveluiden toteutustekniikat ovat kehittyneet jatkuvasti viimeisen kymmenen vuoden aikana. Vanhoja verkkopalvelujärjestelmiä uudistetaan ja kokonaan uudenlaisiin arkkitehtuureihin perustuvia järjestelmiä tuodaan näiden rinnalle tai tilalle. Uusia tekniikoita on arvioitava ja erilaisia prototyyppejä kehitettävä, jotta voidaan löytää oikeat ratkaisut ja säilyttää kilpailukyky nopeasti kehittyvässä ympäristössä.

Viime aikoina ovat huomiota saaneet portaalit, joiden ideana on tuoda yrityksen kaikki palvelut yhtenäisen käyttöliittymän taakse. Portaaleita voidaan toteuttaa ja räätälöidä eri kohdeyleisöille, kuten yrityksen omille työntekijöille, asiakkaille tai yhteistyökumppaneille. Portaalit tarjoavat usein hienojakoisempaakin, käyttäjäkohtaista räätälöintiä eli personointia ja siten mahdollisuuksia paremmin palvella käyttäjiä. Portaaleiden käyttöliittymissä esiintyy sekä pysyvää, harvoin uudistettavaa sisältöä, että dynaamista, tosiaikaisesti käyttäjälle tuotettavaa tai ulkopuolisista järjestelmistä haettavaa sisältöä.

Tässä työssä keskitytään siihen, miten portaalin ulkopuolisten järjestelmien tuottamaa dynaamista tietoa voidaan käsitellä portaalissa ja toteuttaa käyttöliittymän niitä osia, jotka vahvasti hyödyntävät tätä informaatiota. Työssä vertaillaan kahta tähän tarkoitukseen soveltuvaa tekniikkaa toteuttamalla molempien avulla joukko yleisesti esiintyviä käyttöliittymärakenteita ja niiden tuottamiseen liittyvä tiedonkäsittely. Tekniikoita vertaillaan kehittäjän näkökulmasta painottaen toteutustyön suoraviivaisuutta ja kustannustehokkuutta ja toisaalta ajonaikaisen suorituskyvyn näkökulmasta mittaamalla eri ratkaisujen suoritusaikoja. Tavoitteena on pystyä arvioimaan tekniikoiden soveltuvuutta kohdeyrityksessä toteutettavissa verkkopalveluissa käytettäväksi.

Ympäristönä on Java 2 Enterprise Edition (J2EE) –alustalla [44] toimiva portaali, jonka tehtävänä on tuottaa käyttöliittymä. Portaalissa käsiteltävän tiedon esitystapana ja osittain ratkaisuissa käytettävien tekniikoiden perustana on

rakenteisen tiedon kuvaamiseen kehitetty *Extensible Markup Language* (XML) – kieli [12], jota kuvataan luvussa kaksi.

Ensimmäiseksi toteutustekniikaksi valittua XML-dokumenttien käsittelyyn ja muunnoksiin tarkoitettua *Extensible Stylesheet Language Transformations* (XSLT) –kieltä [16] ja sen osaa, *XML Path* –lausekekieltä [17] käsitellään luvussa kolme. Luku neljä kertoo vertailuratkaisussa sovellettavista Java-tekniikoista ja erityisesti käyttöliittymäsivujen toteuttamisessa käytettävästä *JavaServer Pages* (JSP) –tekniikasta [38] ja sen *JSP Standard Tag Library* (JSTL) –laajennuksesta [19].

Luvussa viisi käydään läpi erilaisia yllä mainittujen tekniikoiden soveltamistapoja ja tuodaan esille tärkeitä ratkaisuissa huomioonotettavia seikkoja liittyen esimerkiksi suorituskyvyn parantamiseen. Luku kuusi kuvaa työssä toteutettavien ratkaisujen mahdollisen toimintaympäristön ja työlle asetetut reunaehdot ja vaatimukset.

Toteutetut ratkaisut kuvataan luvussa seitsemän. Työhön valittiin joukko yleisesti esiintyviä käyttöliittymärakenteita sekä suunniteltiin ja toteutettiin niihin liittyvä toiminnallisuus molemmilla vertailuun valituilla tekniikoilla. Luvussa kahdeksan esitetään arviot tekniikoista perustuen kehittämisessä saatuihin kokemuksiin. Lisäksi esitetään suorituskykymittausten tulokset ja analysoidaan niitä. Lopuksi esitetään yhteenveto koko työstä ja pohditaan joitakin jatkotutkimusmahdollisuuksia luvussa yhdeksän.

2. Luku - Extensible Markup Language

"The primary motivation behind the development of XML was the recognition that maintaining content for the web in HTML was a really bad idea, because it failed to separate content from presentation, and because HTML browsers were full of proprietary features that inhibited interoperability."

- Michael Kay

Tässä luvussa kuvataan lyhyesti XML-kieltä ja siihen liittyviä perustekniikoita.

2.1 Yleistä

Extensible Markup Language (XML) on laajennettava, rakenteisen tiedon kuvauskieli. XML-standardi määrittelee ainoastaan yleisen teknisen perusrakenteen erilaisten XML-pohjaisten kielten rungoksi. Tässä mielessä XML on metakieli; kieli uusien kielten kuvaamiseen.

XML:n standardoinnista vastaa World Wide Web Consortium (W3C). Standardointityö jakaantuu useisiin eri työryhmiin. Ydinstandardeista liittyen itse kielen rakenteeseen, nimiavaruuksiin (XML Namespace) [11] ja tietomalliin (XML Information Set) [18] vastaa XML Core Working Group –työryhmä. Muista työryhmistä mainittakoon XSL Working Group, jonka tehtäväkenttään kuuluu XML-dokumenttien muunnoksiin käytettävän Extensible Stylesheet Language –kieliperheen (XSL) määrittely [40]. Tällä hetkellä julkaistuja versioita XML-kielestä on kaksi; 1.0 ja 1.1. Tässä työssä tukeudutaan versioon 1.0.

XML kehitettiin alun perin ratkaisemaan HTML-pohjaisissa verkkopalveluissa havaittuja ongelmia. Haluttiin selkeästi erottaa tietosisältö ja sen esitystapa toisistaan, mihin HTML ei kyennyt. Lisäksi HTML:n käyttö oli ja on turhan kirjavaa valmistajakohtaisten tulkintojen ja ratkaisujen vuoksi [26].

Tarvittiin tarkasti määritelty syntaksi sekä tiedon että sen esitystavan kuvaamiseen. Pohjaksi otettiin olemassaoleva standardi, Standard Generalized Markup Language (SGML), johon myös HTML perustuu. SGML:n monimutkaisuus tiedostettiin ja XML pyrittiin määrittelemään mahdollisimman yksinkertaiseksi ja pieneksi osajoukoksi ko. standardista [26].

XML 1.0 –standardin uusin versio (Third Edition) määrittelee kymmenen päämäärää XML:n suunnittelulle. Näissä korostuu erityisesti käytön helppous, käytettävyys laajassa kirjossa sovelluksia ja se, että XML:n on oltava syntaksiltaan yksinkertainen ja ihmisen ymmärrettävissä sellaisenaan [12].

2.2 XML-dokumentit

XML-dokumentti on looginen tietorakenne, joka voi koostua yhdestä tiedostosta tai olla hajautettuna fyysisesti useaan eri paikkaan. XML-dokumentti on tietovarasto, sillä se pitää sisällään, muotoilee, nimeää ja antaa tiedolle rakenteen [41]. XML-dokumentilla on sekä fyysinen, että looginen rakenne. XML-jäsennin muodostaa dokumentin fyysiseen rakenteeseen perustuen dokumentin sisällöstä loogisen rakenteen sovellusohjelmien käsiteltäväksi.

Merkkijonotiedosto on (hyvinmuotoiltu, well-formed) XML-dokumentti, jos se noudattaa tiettyjä XML-standardissa määriteltyjä syntaktisia perussääntöjä [12]. XML-dokumentti on lisäksi validi (valid), jos se noudattaa dokumentissa viitattuun tai dokumenttiin sisältyvään Document Type Definition (DTD) – skeemaan kirjattuja elementtejä ja attribuutteja koskevia rajoitteita.

2.2.1 Fyysinen rakenne

Fyysisesti XML-dokumentti koostuu entiteeteistä (entity) [12]. Entiteetit ovat joko jäsennettävää (parsed) tai jäsentämätöntä (unparsed) tietoa. Jäsennettävä tieto on merkkijonotietoa (character data) tai merkintäkieltä (markup). Merkkijonotieto on dokumentin varsinaista sisältöä ja merkintäkielellä määritellään tälle tiedolle looginen rakenne.

Alla on esimerkki XML-dokumentista, jossa varsinaista sisältöä on ainoastaan **lihavoitu** teksti sekä välilyönnit ja rivinvaihdot. Kaikki *kursiivit* merkkijonot ovat merkintäkieltä:

```
<?xml version="1.0" encoding="UTF-8"?>
<esimerkki>
  <sisältö>
    merkkijonosisältöä
  </sisältö>
</esimerkki>
```

Fyysisesti XML-dokumentti ei välttämättä ole yksi tiedosto vaan voi koostua eri paikoissa sijaitsevista osista. Myös tällaiset ulkoiset osat ovat entiteettejä ja niitä voidaan sisällyttää dokumenttiin *entiteettiviittauksilla* (entity reference), joita kuvataan alla tarkemmin.

2.2.2 Prologi ja XML-deklaraatio

XML-dokumentin alussa on valinnainen prologi. Prologin pitäisi alkaa XML-deklaraatiolla [12]. XML-deklaraatiossa määritellään yleensä käytettävä versio ja dokumentin merkkikoodaus, mutta myös muita tietoja voidaan kuvata. XML-deklaraation on sijaittava dokumentissa ensimmäisenä, ennen dokumentin juurielementtiä. Seuraavassa esimerkissä on XML-deklaraatio:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Jos dokumentissa ei ole määritelty merkistön koodausta, sen oletetaan olevan UTF-8 (tai UTF-16, mikäli dokumentti alkaa merkillä xFEFF) [41].

XML-dokumentin prologissa voidaan määritellä myös dokumentin tyyppi viittaamalla erilliseen dokumentin rakenteen kuvaukseen (Document Type Definition, DTD) [41]. DTD määrittelee dokumentille rakenteen kuvaamalla millaisista elementeistä ja attribuuteista se voi koostua. DTD:ssa voi olla myös *entiteettideklaraatioita*. XML-dokumentissa olevilla entiteettiviitteillä viitataan entiteettideklaraatioissa määriteltyihin entiteetteihin.

2.2.3 Entiteeteistä

Entiteetit yleisesti ovat XML-dokumenttiin liitettäviä merkkijonoja tai XML-dokumentteja tai viittauksella liitettäviä merkkijonomuotoisia tai binaarisia tiedostoja. Yleisiä (general) entiteettejä käytetään XML-dokumentin sisällössä. Parametrientiteettejä (parameter entity) käytetään DTD-määrittelyissä [41].

Yleiset entiteetit jakauvat kahteen ryhmään. Jäsennetty (parsed) entiteetti on XML-muotoista merkkijonosisältöä, joka näkyy sovellukselle korvattuna entiteettimäärittelyn (entity declaration) sisällöllä. Jäsentämätön (unparsed) entiteetti voi olla mitä tahansa sisältöä, ja se näkyy sovellukselle vain viitteenä.

Sisäinen (internal) entiteetti on jäsennetty merkkijono, jonka määrittely sisältyy kokonaisuudessaan itse XML-dokumenttiin. Ulkoinen (external) entiteetti on jäsennetty tai jäsentämätön ja sen määrittely on erillisessä dokumentissa.

Entiteetteihin ja normaalissa merkkijonosisällössä kiellettyihin Unicode-merkkeihin viitataan alla olevassa esimerkissä kuvatulla tavalla. Tässä työssä relevantteja ovat vain jäsennetyt, sisäiset entiteetit ja XML:n sisäänrakennetut entiteetit. Yksittäisiin merkkeihin (character entity) voidaan viitata myös joko 10- tai 16-järjestelmän mukaisella numeroarvolla. Mihin tahansa merkkiin on mahdollista viitata näin.

```
XML:n sisäänrakennetut entiteetit    &amp; &lt; &gt; &apos; &quot;
.. näkyvät sovellukselle merkkeinä   &     <    >    '      "

Merkki   10-järjestelmässä on sama kuin   heksadesimaalina
```

Alla olevassa esimerkissä on määritelty kaksi sisäistä, jäsennettyä entiteettiä. Jäsennettäessä tätä dokumentti XML-parserilla, näkyisi `dokumentti`-elementin sisältö sovellukselle merkkijonona "€ & © omina entiteetteinä".

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE dokumentti [
  <!ENTITY euro "&#x80;">
  <!ENTITY copy "&#xA9;">
]>
<dokumentti>
  &euro; &amp; &copy; omina entiteetteinä
</dokumentti>
```

2.2.4 Elementit ja tagit

XML-dokumentin looginen rakenteellinen perusyksikkö on *elementti* [41]. Elementin alku ja loppu merkitään *tageilla.* Aloittava tagi on < (pienempi-kuin) ja > (suurempi-kuin) –merkeillä ympäröity merkkijono. Lopettavassa tagissa on lisäksi ennen merkkijonoa kauttaviiva (/). Elementit voivat sisältää toisia elementtejä ja muodostaa näin monitasoisen hierarkian. Lisäksi ne voivat sisältää merkkijonotietoa, kommentteja, attribuutteja, CDATA-sektioita ja prosessointiohjeita. Elementti voi olla myös tyhjä. Hierarkiassa ulointa elementtiä kutsutaan juuri- tai dokumenttielementiksi. Seuraavassa esimerkissä havainnollistetaan erilaisia mahdollisuuksia muodostaa loogisia rakenteita elementeillä:

```
<?xml version="1.0" encoding="UTF-8">
<juurielementti>
  <tyhja_elementti />
  <tyhja_elementti></tyhja_elementti>
  <taulukkoelementti>
    <sisaltoelementti>tietosisältöä</sisaltoelementti>
    <sisaltoelementti>lisää tietosisältöä </sisaltoelementti>
  </taulukkoelementti>
</juurielementti>
```

Esimerkin mukaisesti tyhjä elementti ei tarvitse erillistä lopettavaa tagia.

2.2.5 Attribuutit

Attribuutti on nimi-arvo-pari, joka määrittää ominaisuuksia tietylle elementille tai sen sisällölle [41]. Attribuuteilla voidaan yksinkertaisesti antaa elementille jokin tunniste, sijoittaa se johonkin kategoriaan tai liittää siihen mitä tahansa tilanteeseen sopivaa *metatietoa.* XML-dokumenteissa on usein samannimisiä elementtejä samassa kontekstissa, jolloin attribuutit istuvat luontevasti erottamaan näitä elementtejä toisistaan. Seuraavassa esimerkissä kuvataan attribuuttien käyttöä:

```
<virhe avain="v2" xml:lang="fi">Tuli virhe 2.</virhe>
<virhe avain="v2" xml:lang="en">Error 2 occurred.</virhe>
```

XML-standardi määrittelee joitakin attribuutteja erityiskäyttöön. Esimerkiksi `xml:lang` –attribuutilla voidaan määritellä RFC 3066:n mukainen kieli tai `id` – attribuutilla dokumentin sisällä yksikäsitteinen avain elementille [12].

2.2.6 Prosessointiohjeet

Prosessointiohjeet ovat sovellukselle tarkoitettuja ohjeita dokumentin käsittelyyn [12]. Ne koostuvat kahdesta osasta; kohteesta (`target`) ja kohteelle välitettävästi tiedosta (`data`) [41]. XML-jäsennin välittää prosessointiohjeet sovellusohjelmalle, joka voi kohteen tunnistaessaan hyödyntää prosessointiohjetta haluamallaan tavalla tai hylätä sen. Prosessointiohjeiden syntaksi on seuraavanlainen, missä `data` voi olla mitä tahansa XML:ssä yleensä sallittua merkkijonotietoa:

```
<?target data?>
```

2.2.7 Kommentit

Kommentit ovat muistiinpanoja, joita XML-jäsentimen *ei tarvitse* välittää sovellusohjelmalle [41]. Kommenteilla välitetään tietoa XML-dokumentteja käsitteleville ihmisille, ei sovelluksille. Kommentit voivat sijaita dokumentissa missä tahansa paitsi ennen XML-deklaraatiota tai tagien sisällä.

```
<!-- tämä on kommentti -->
```

2.2.8 CDATA-sektiot

CDATA on lyhenne sanoista *character data*, eli viittaa sisältöön, joka ei ole merkintäkieltä [41]. CDATA-sektiot tulkitaan normaalina tekstinä ja voivat siten sisältää elementtien sisällössä kiellettyjä merkkejä. Ainoastaan CDATA-sektion lopettava merkkijono "]]>" on kielletty sen sisältönä.

```
<elementti>
  <![CDATA[
    jäsentämätöntä merkkijonotietoa joten < & ja > ovat tässä
    sallittuja
  ]]>
</elementti>
```

2.2.9 Merkistöt ja merkkikoodaukset

XML-dokumentin merkistö on ISO/IEC 10646:2000 –standardin mukainen [12]. Tämä Universal Multiple-Octet Coded Character Set (UCS) on 31-bittinen ja sisältää siten nimiavaruutta yli kahdelle miljardille merkille. Vastaava Unicode Consortiumin standardi Unicode on 16-bittinen [41]. Unicode on osajoukko UCS:stä, joten käytännössä voidaan puhua Unicode-merkistöstä XML-dokumenttien yhteydessä. Lisäksi useimmiten riittää ISO-Latin-1-merkistö (ISO-8859-1) ja se muodostaakin Unicode-merkistön 256 ensimmäistä merkkiä.

Unicode-merkistö koodataan yleisimmin UTF-8-muotoon (UCS Transformation Format for 8-bits). Tämä on oletuskoodaus, mikäli mitään koodausta ei ole erikseen määritelty XML-deklaraatiossa [41]. XML-jäsentimen on standardin mukaan kuitenkin tuettava vähintään UTF-8 ja UTF-16 –koodauksia. UTF-8-koodattuna merkit 0-127 (US-ASCII) ilmaistaan yhdellä tavulla. Kaikki muut Unicode-merkit ilmaistaan kahdesta kuuteen tavulla.

2.2.10 Nimiavaruudet

XML-nimiavaruudet tarjoavat keinon määritellä XML-dokumenteissa esiintyville elementeille ja attribuuteille yksikäsitteiset nimiavaruudet [11]. Nimiavaruudet mahdollistavat samannimisten, mutta eri merkityksessä olevien elementtien esiintymisen samassa dokumentissa. Sovellusohjelmat voivat nimiavaruuksien avulla erottaa dokumentista haluamansa elementit ja attribuutit.

XML-nimiavaruuden määrittää tietty URI-viite (Universal Resource Identifier) [11][8]. Eri URI-viitteet käsitetään samaksi vain, jos ne ovat merkki merkiltä täsmälleen samanlaiset. Nimiavaruus liitetään yleensä johonkin etuliitteeseen, jota käytetään dokumentissa niiden elementtien yhteydessä, joiden halutaan kuuluvan ko. nimiavaruuteen. Nimiavaruus voidaan määritellä myös nimettömäksi

(oletusarvoiseksi), jolloin sen kanssa ei käytetä mitään etuliitettä. Tällöin kaikki nimiavaruuden määrittelelevän elementin sisällä olevat elementit kuuluvat samaan nimettömään nimiavaruuteeteen, elleivät ne sisällä erillisiä nimiavaruusmäärittelyjä.

Alla olevassa esimerkissä `nimetty:elementti` kuuluu nimiavaruuteen `http://nimetty/`. `elementti` ja kaikki muutkin ilman etuliitettä nimetyt *kursiivit* elementit kuuluvat nimiavaruuteen `http://oletus/`:

```
<?xml version="1.0"?>
<juuri xmlns:nimetty="http://nimetty/"
       xmlns="http://oletus/">
  <nimetty:elementti>
    <sisainen> kuuluu nimiavaruuteen http://oletus/ </sisainen>
  </nimetty:elementti>
  <elementti />
</juuri>
```

2.3 XML-dokumenttien käsittely

2.3.1 SAX ja DOM

Simple API for XML (SAX) on tapahtumapohjainen (event-driven) rajapinta XML-dokumenttien käsittelyyn [13]. Alun perin se kehitettiin xml-dev-postituslistalla käytävien keskustelujen pohjalta ja on vapaasti kaikkien käytettävissä (public domain). Tapahtumapohjaisuus tässä yhteydessä tarkoittaa, että sovellusohjelman on rekisteröitävä XML-dokumenttia lukevalle komponentille haluamansa käsittelijät, joita kutsutaan törmättäessä XML-dokumentissa esiintyviin erilaisiin rakenteellisiin elementteihin. Tällainen tapahtumapohjainen rajapinta on hyvin yksinkertainen toteuttaa ja soveltuu tilanteisiin, joissa dokumentin sisältö käydään läpi vain kertaalleen eikä siihen tehdä satunnaisia hakuja tai päivityksiä. Erityisesti, jos sisältö tallennetaan joka tapauksessa toiseen, sovelluskohtaiseen muotoon, on SAX oikea valinta XML:n käsittelyyn. SAX-rajapinnasta on olemassa kaksi pääversiota – 1 ja 2 – joista jälkimmäinen on käytännön standardi.

Document Object Model (DOM) on World-Wide Web Consortiumin (W3C) määrittelemä ympäristö- ja kieliriippumaton rajapinta, joka tarjoaa välineet dynaamisesti lukea ja päivittää erilaisten dokumenttien sisältöä, rakennetta ja tyyliä [29]. DOM on dokumentista luotu muistinvarainen puurakenne ja sisältää koko käsiteltävän dokumentin sisällön olioina. DOM soveltuu käytettäväksi tilanteissa, joissa dokumenttiin tehdään satunnaisia hakuja ja mahdollisesti päivityksiä.

2.3.2 Validointi

XML-dokumenttien jäsentämiseen voidaan liittää validointi. Tällöin XML-jäsentäjä lukee ensin tietyt dokumentin rakennetta koskevat säännöt ja varsinaista dokumenttia jäsentäessään tarkistaa, että dokumentti noudattaa kyseisiä sääntöjä [41]. Jäsennin tuottaa tällöin vähintään tiedon siitä, oliko jäsennettävä dokumentti validi mutta voi tuottaa myös laajemman tietojoukon dokumentin sisällöstä. Tämä

Post Schema Validation Infoset (PSVI) sisältää tarkempaa tietoa dokumentin rakenteesta ja elementtien ja attribuuttien tietotyypeistä.

XML-suosituksen mukaan dokumentti on validi noudattaessaan nimenomaan DTD-määrittelyn sääntöjä, mutta käytännössä usein käytetään XML Schema- tai muita skeema-määrittelyitä dokumenttien validointiin.

Validoinnissa voidaan huomioida seuraavia asioita [41]:

- rakenne: elementtien ja attribuuttien sijoittuminen dokumentissa
- tietotyypit: merkkijonotiedon muoto (esim. desimaaliluvut ja päivämäärät)
- eheys: dokumentissa olevien sisäisten ja ulkoisten linkkien toimivuus
- liiketoimintasäännöt: tarkempi tekstisisällön analysointi

2.3.3 DTD ja XML Schema

Vanhin ja yleisimmin tuettu skeemakieli on Document Type Definition (DTD) [41]. DTD määrittää XML-dokumentille sallitut elementit ja niiden muodostaman rakenteen. Tietyn elementin sisällä sallitut ja vaaditut elementit, niiden järjestys ja lukumäärä sekä mahdollisuus sisällyttää myös merkkijonotietoa kuuluvat tähän rakenteen määrittelyyn. Lisäksi määritellään elementeissä sallitut ja vaaditut attribuutit sekä niiden mahdolliset oletusarvot ja sallitut arvojoukot.

DTD:n mahdollisuudet ovat varsin niukat. W3C kehittikin XML Schema-kielen korvaamaan DTD-määrittelyt XML-dokumenttien rakenteen määrittelyssä. XML Schemojen käyttö tuo joitakin etuja verrattuna DTD-dokumentteihin [41]. Nimiavaruuksien käyttö on mahdollista ja elementtien järjestyksen ja lukumäärän määrittelyssä on enemmän ilmaisuvoimaa. Elementtien ja attribuuttien merkkijonosisällölle voidaan määritellä tietotyyppejä käyttäen joko valmiita XML Schema-tietotyyppejä tai laajentamalla niitä säännöllisiin lausekkeisiin perustuvalla mekanismilla.

2.3.4 XML Information Set

XML Information Set (InfoSet) määrittelee erään tietomallin XML-dokumenteille [18]. Se kuvaa joukon ominaisuuksia kullekin XML-dokumenteissa esiintyvälle rakenteelliselle osalle ja on suunniteltu tukemaan muita määrittelyjä. XML-dokumenttien käsittelyyn erikoistuneet tekniikat, kuten XPath, XSLT ja DOM, määrittelevät omat, kyseisten tekniikoiden erityistarpeiden mukaan suunnitellut tietomallinsa. Koska tässä työssä käsitellään XML-dokumentteja XPath- ja XSLT-tekniikoiden avulla, ei InfoSet-määrittelyä tässä tarkemmin kuvata.

3. Luku - XML-muunnostekniikat

3.1 Yleistä

SGML-pohjaisilla kielillä – kuten HTML ja XML – tehtyjen dokumenttien käsittelyyn on kehitetty useita eri ratkaisuja. World-Wide Web Consortium (W3C) julkaisi vuonna 1996 Document Style Semantics and Specification Language (DSSSL) –kielen, joka on ollut perustana uusimmille ratkaisuille [41]. DSSSL itse oli laajahko ja monimutkainen Scheme-kieleen perustuva ohjelmointikieli, eikä siten kovinkaan hyvä käytännön sovelluksiin. Tarvittiin yksinkertaisempia ratkaisuja ja näin syntyi XML dokumenttien kuvaamiseen ja XSL-kieliperhe dokumenttien muunnoksiin ja ulkoasun kuvaamiseen.

Extensible Stylesheet Language (XSL) on kieliperhe, joka koostuu kolmesta osasta [40]. XSL Transformations (XSLT) on funktionaalinen ohjelmointikieli XML-dokumenttien muunnoksiin. Toinen osa, XML Path Language (XPath) on lausekekieli, jota käytetään osana eri tekniikoita XML-dokumentin eri osiin viittaamiseen. Kolmas osa, XSL Formatting Objects (XSL-FO) on muotoilukieli, jolla määritellään dokumenteille tarkka ulkoasu eri esitystapoja varten.

Tässä keskitytään Extensible Stylesheet Transformations (XSLT) –tekniikkaan. Sovellettavat versiot ovat XSLT 1.0 ja XPath 1.0. Tätä kirjoitettaessa ovat standardoinnin viimeisessä vaiheessa toisiinsa liittyvät XPath 2.0, XSLT 2.0 ja XQuery 1.0. Näistäkin kerrotaan lyhyesti, mutta tekniikoita ei sovelleta tässä työssä.

3.2 XML Path Language

XML Path Language (XPath) on kieli XML-dokumentin eri osiin viittaamiseen [17]. Lisäksi XPath sisältää yksinkertaisia työkaluja merkkijonojen, numeroiden ja Boolen-arvojen käsittelyyn. XPathilla voidaan viittaamisen lisäksi tutkia, noudattaako viitattu XML-dokumentin osa tiettyjä ehtoja. XPath käsittelee XML-dokumentin loogista rakennetta ja määrittelee oman tietomallinsa tätä varten. Tekniikka on siten riippumaton varsinaisesta XML-syntaksista. XPath 1.0 tai sen jokin osajoukko on käytössä ainakin XSLT ja XPointer -tekniikoissa ja lisäksi sitä voidaan käyttää näistä riippumattomasti.

3.2.1 Tietomalli käsiteltävälle dokumentille

XPath mallintaa XML-dokumenttia solmujen muodostamana puurakenteena [22]. Erilaisia solmutyyppejä on seitsemän:

- juurisolmu (root, document node): XML-dokumentin alkupiste, ennen mitään sisältöä. Juurella on lapsena täsmälleen yksi elementti – XML-dokumentin dokumenttielementti. Lisäksi juurisolmulla voi olla lapsina kommentti- ja prosessointiohjesolmuja.
- elementtisolmu (element node): XML-dokumentin elementti
- attribuuttisolmu (attribute node): XML-elementin attribuutti
- tekstisolmu (text node): merkkijono elementin sisällä. Myös CDATA-sektiot ovat XPathin kannalta tekstisolmuja.
- nimiavaruussolmu (namespace node): nimiavaruusmäärittelyn etuliite ja URI-osoite
- kommenttisolmu (comment node): XML-kommentin sisältö
- prosessointiohjesolmu (prosessing instruction node): XML-prosessointiohje

XML-dokumentista muodostettu puu noudattaa osittain ns. dokumenttijärjestystä, eli sitä järjestystä, missä solmut esiintyvät dokumentissa [17]. Juurisolmu on aina ensimmäisenä. Elementti- ja tekstisolmut ovat puussa samassa järjestyksessä kuin dokumentissa. Attribuutti- ja nimiavaruussolmut ovat elementtisolmun sisällä ennen muita lapsisolmuja. Kuitenkin attribuuttisolmujen samoin kuin nimiavaruussolmujen sisäinen järjestys on toteutuksesta riippuva.

3.2.2 Lausekkeet, polut ja konteksti

XPath-tekniikan perusrakenne on lauseke (expression) [17]. Lausekkeen arvo voi olla neljää eri perustyyppiä. Solmujoukko (node-set) on järjestämätön joukko yllä kuvatun tietomallin mukaisia solmuja ilman duplikaatteja. Muut mahdolliset arvot ovat Boolen-arvo (boolean), liukuluku (number) ja UCS-merkistön merkkejä sisältävä merkkijono (string).

XPath-lausekkeen evaluoinnilla on aina jokin konteksti, joka koostuu seuraavista tiedoista [17]:

- kontekstisolmu eli käsiteltävänä oleva solmu
- kontekstisolmun sijainti käsiteltävässä solmujoukossa (1-n) ja kyseisen solmujoukon koko (n)
- joukko muuttujia ja niihin liitettyjä arvoja, joiden tyypitys vastaa lausekkeiden paluuarvojen tyypitystä
- funktiokirjasto

- joukko XML-nimiavaruusmäärittelyitä, jotka ovat kontekstisolmun näkyvyysalueella. Nimiavaruusmäärittely tässä yhteydessä on kuvaus etuliitteestä nimiavaruuden määrittävään URI-tunnisteeseen.

Tärkein lauseketyyppi on polku (location path) [22]. XPath-polulla voidaan viitata yhteen tai useampaan solmuun XPath-tietomallia noudattavassa puurakenteessa. Polku on suhteessa kontekstisolmuun mutta se voidaan määrittää myös alkaen dokumentin juurisolmusta "/". Polku koostuu askeleista (location step), jotka erotetaan kautta-viivalla (/). Kukin askel koostuu kolmesta osasta [17]:

```
akseli::solmutesti[predikaatti]
```

Akseli (axis) kuvaa valittavien solmujen suhdetta kontekstisolmuun puurakenteessa. Akseleita on 13, joista esimerkkeinä mainittakoon `child` (solmun lapsielementit), `attribute` (solmun attribuutit) ja `self` (solmu itse) [17]. Solmutesti (node test) on solmun tyypin, nimen tai molemmat määrittävä merkkijono. Predikaatti (predicate) on valinnainen Boolen-arvoinen lauseke, joka rajaa valittavia solmuja tarkemmin. Jos lauseke ei ole suoraan Boolen-arvoinen, se muunnetaan sellaiseksi tiettyjen sääntöjen mukaan [17]. Predikaatissa voidaan käyttää kaikkia XPath-standardiin kuuluvia operaattoreita ja funktioita.

XPath-polut voidaan kirjoittaa lyhentämättömässä (unabbreviated) tai lyhennetyssä (abbreviated) muodossa. Alla on kahden askeleen mittainen XPath-polku, joka valitsee ensin kaikki kontekstielementin lapsielementit ja sitten ko. elementeistä sellaiset lapsielementit, joilla on attribuutin `attribuutti` arvona `arvo`.

```
child::*/child::*[attribute::attribuutti = 'arvo']

*/*[@attribuutti = 'arvo']
```

Kertomerkki (*) valitsee lapsina olevat elementit (`child`-akseli). *At*-merkki (@) viittaa attribuutteihin (`attribute`-akseliin).

Solmuja valitsevat polut ovat vain yksi XPath-lausekkeiden erikoistapaus. Lausekkeilla voidaan lisäksi laskea aritmeettisia laskutoimituksia, evaluoida loogisia lausekkeita Boolen-arvojen avulla, viitata ulkoisiin muuttujiin ja kutsua funktioita [22].

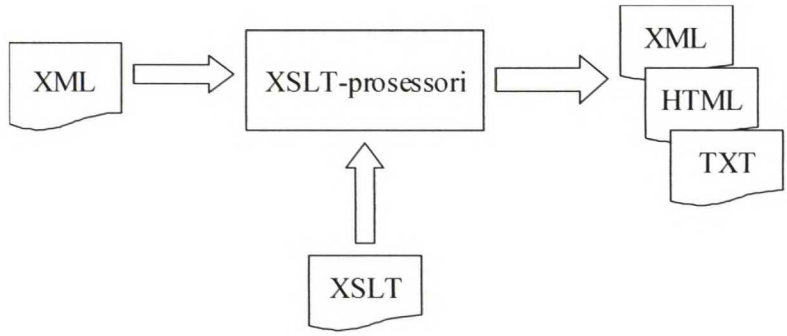
XPath 1.0 –suositus määrittelee 27 pakollista funktiota [17]. Näillä voidaan käsitellä numeroita, merkkijonoja, Boolen-arvoja ja solmujoukkoja (node-set). Solmutestit ovat funktioiden Boolen-arvoja palauttava erikoistapaus. `text()` on tosi, jos kontekstisolmu on tekstimuotoinen. `node()` on tosi mille tahansa solmulle, paitsi attribuuteille. `comment()` ja `processing-instruction()` toimivat nimiensä mukaisesti.

3.3 Extensible Stylesheet Language – Transformations

3.3.1 Yleistä

XSLT-prosessori (muunnosprosessori, XSLT-processor) on ohjelmisto tai laite, joka ottaa syötteenä XSLT-muunnosdokumentin ja käsiteltävän XML-dokumentin

[41]. XSLT-prosessori käsittelee XML-dokumentista muodostetun lähdepuun (source tree) XSLT-dokumentissa olevien deklaratiivisten sääntöjen mukaisesti ja tuottaa (yleensä) toisen XML-dokumentin, jota kutsutaan kohdepuuksi (result tree). Lähdepuun solmut ja tietotyypit noudattavat XPath-tietomallia.



Kuva 3-1: XSLT-muunnos

Muunnos tapahtuu liittämällä lähdepuun solmuja hahmoilla (pattern) malleihin (template), jotka käsittelevät ko. solmut [16]. Lähdepuuta käydään läpi dokumenttijärjestyksessä ja XSLT-dokumentista etsitään sopivia hahmoja lähdepuussa vastaan tuleville solmuille. Kullekin käsiteltävälle solmulle instantioidaan se malli, johon liittyvä hahmo täsmää (match) solmuun. Malli tuottaa haluttua sisältöä kohdepuuhun ja mahdollisesti jatkaa prosessointia rekursiivisesti lähdepuun muille solmuille.

Alla olevassa esimerkissä on kokonainen XSLT-dokumentti, jossa dokumentin juureen täsmäävä malli käsittelee koko lähdedokumentin:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" encoding="UTF-8" />

  <xsl:template match="/">
    Tämä merkkijono menee tulospuuhun ja mitään muuta ei tulosteta.
  </xsl:template>
</xsl:stylesheet>
```

XSLT-hahmo (pattern) on XSLT-suosituksen määrittämä osajoukko XPath-lausekkeista [22]. Yllä olevassa esimerkissä on yksi hahmo `xsl:template`-elementin `match`-attribuutin sisältönä. Hahmon käsittelyn lopputulos voi olla ainoastaan solmujoukko. Hahmo voi kuitenkin sisäisesti käyttää mitä tahansa XPath-funktioita ja solmutestejä ja lisäksi yhdeksää XSLT-funktiota.

XSLT käyttää funktioiden ja XPath-lausekkeiden paluuarvoina samoja tietotyyppejä kuin XPath (solmujoukko, Boolen-arvo, numero ja merkkijono) lisäten kuitenkin yhden uuden [22]. Tämä *result tree fragment* on kohdepuuhun tulostettavaa sisältöä ja sen ei ole pakko olla XML-tyyppistä. Kohdepuun palasia syntyy mallien (template) instantioinnissa ja niitä käsitellään merkkijonoina. Siten kohdepuun palasiin ei voida soveltaa lisäkäsittelyä esimerkiksi XPath-polkujen avulla ilman valmistajakohtaisia laajennuksia.

XSLT käyttää omaa nimiavaruuttaan erottamaan XSLT-kieliset elementit muista XSLT-dokumentin hahmoissa olevista, tulostettaviksi tarkoitetuista elementeistä

[16]. XSLT-nimiavaruuden URI on `http://www.w3.org/1999/XSL/Transform` ja etuliitteeksi on vakiintunut `xsl`.

3.3.2 XSLT-dokumentti – xsl:stylesheet

XSLT-dokumentin juurielementti on `xsl:stylesheet` [16]. Se sisältää pakollisena attribuuttina käytettävän version, joka tässä tapauksessa on `1.0`. Valinnaisina attribuutteina on usein erilaisia nimiavaruusmäärittelyitä. Suoraan `xsl:stylesheet`-elementin sisällä olevia elementtejä kutsutaan päätason elementeiksi ja niitä kuvataan lyhyesti alla.

Tärkein elementti on `xsl:template` eli malli, joita on yleensä useita ja joissa varsinainen XML-käsittely ja tulostaminen tapahtuu. Tulospuun sarjallistamiseen ja lopputuloksen syntaksiin voidaan vaikuttaa `xsl:output`-elementin attribuuteilla. Erilaisten usein viitattavien rakenteiden käyttöä on mahdollista tehostaa luomalla muunnoskohtainen hajautusrakenne valituista solmuista `xsl:key`-elementillä.

XSLT-muunnoksille voidaan määrittää juuritasolla vakioarvoisia muuttujia. Näitä kuvataan `xsl:variable`-elementillä. Jos muunnokselle halutaan välittää parametreja käsiteltävän dokumentin lisäksi, voidaan tähän käyttää `xsl:param`-elementtejä.

`xsl:import` ja `xsl:include` –elementeillä voidaan sisällyttää malleja muista tiedostoista ja siten modularisoida toteutusta. Vastaavasti kehittämisessä auttaa `xsl:attribute-set`, jolla kertaalleen määriteltyyn attribuuttijoukkoon voidaan viitata muualla dokumentissa.

Lisäksi whitespace-merkkien käsittelyä voidaan hallita määrittämällä elementit, joiden sisältämät whitespace-merkit halutaan poistaa ennen muunnosta tai vastaavasti ottaa ne kaikki muunnokseen mukaan. Nimiavaruuksille voidaan määrittää kuvauksia, joiden seurauksena tietty XML-nimiavaruus kuvautuu automaattisesti joksikin toiseksi nimiavaruudeksi tulospuuhun. Numeroiden lokalisoitu muotoilu on osa XSLT:n perustoiminnallisuutta ja siihen liittyen määritellään nimettyjä muotoilusääntöjä.

3.3.3 Malli – xsl:template

Pääasiassa XSLT-dokumentti on kokoelma malleja (template) [41]. Mallit käsittelevät lähdepuun solmuja ja ohjaavat käsittelyä rekursiivisesti eteenpäin näin haluttaessa. Mallit täsmäävät tiettyihin solmuihin `match`-attribuutissa määritellyn XPath-hahmon avulla. Alla oleva esimerkki sisältää mallin, joka täsmää HTML-dokumentin otsikon kuvaavaan elementtiin ja tulostaa sen tulospuuhun:

```
<xsl:template match="/html/head/title">
  <xsl:value-of "." />
</xsl:template>
```

Instantioitavalla mallilla on kontekstinaan *nykyinen solmu* (current node) ja *nykyisten solmujen lista* (current node list) [16]. Malli voi hyödyntää tai olla hyödyntämättä näitä tietoja. XPath-lausekkeissa käytettävissä olevaan kontekstiin

sisältyy luonnollisesti XPath-tekniikan yleinen konteksti. Malli voi kutsua rekursiivisesti toisia malleja.

Mikäli rekursiivinen kutsu tehdään `xsl:apply-templates`-elementillä, näkee kutsuttava malli erilaisen kontekstin [16]. Tällöin kontekstisolmuna on joko kutsujan kontekstin lapsisolmu tai kutsujan `xsl:apply-templates`-elementin `select`-attribuutilla erikseen valitsema solmu. Alla olevassa esimerkissä valitaan ensimmäisellä mallissa solmu, jonka sisältö tulostetetaan toisella mallilla:

```
<xsl:template match="/html/head">
  <xsl:apply-templates select="title" />
</xsl:template>

<xsl:template match="title">
  HTML-dokumentin otsikko oli <xsl:value-of "." />
</xsl:template>
```

Kutsuttaessa malleja `xsl:call-template`-elementillä, valitaan instantioitava malli suoraan nimellä ja kontekstisolmu ei vaihdu lainkaan [22]. Alla oleva esimerkki käyttää tätä mekanismia:

```
<xsl:template match="/html/head">
  <xsl:call-template name="otsikon_tulostus" />
</xsl:template>

<xsl:template name="otsikon_tulostus">
  HTML-dokumentin otsikko oli <xsl:value-of "title" />
</xsl:template>
```

Malleille voidaan välittää parametreja käyttämällä kutsun yhteydessä `xsl:with-param`-elementtiä [22]. Mekanismi toimii molemmilla yllä kuvatuilla kutsutavoilla. Vastaanotettavat parametrit määritellään vastaanottavassa mallissa erikseen ja niille voidaan asettaa oletusarvo siltä varalta, ettei parametreja välitetä. Esimerkissä on myös viittaus parametriin $-merkin avulla. Samalla tavoin voidaan viitata XSLT-dokumentin päätasolla määriteltyihin parametreihin ja muuttujiin:

```
<xsl:variable name="muuttuja" select="'päätason muuttujan arvo'" />

<xsl:template match="*">
  <xsl:call-template name="parametroitu_malli">
    <xsl:with-param name="parametri" select="'annettu arvo'" />
  </xsl:call-template>
</xsl:template>

<xsl:template name="parametroitu_malli">
  <xsl:param name="parametri" select="'oletusarvo'" />
  Parametrin arvo on <xsl:value-of select="$parametri" />
  Päätason muuttuja arvo on <xsl:value-of select="$muuttuja" />
</xsl:template>
```

3.3.4 Sisäänrakennetut mallit

XSLT-prosessorilla on aina käytössä joukko sisäänrakennettuja malleja [16]. Näitä instantioidaan silloin, kun mikään erikseen määritellyistä malleista ei täsmää käsiteltävään solmuun. Elementeille ja juurisolmulle sisäänrakennettu malli jatkaa rekursiivisesti käsittelyä. Tämä toimii myös eri moodeissa oleville malleille siten, että moodi säilyy samana rekursiivisissa kutsuissa. Attribuuttien ja

tekstisolmujen sisäänrakennettu malli kopioi sisällön sellaisenaan tulospuuhun. Kommentit ja prosessointiohjeet jätetään käsittelemättä.

3.3.5 Instantioitavan mallin valinta

Usea malli voi joissakin tilanteissa täsmätä samaan solmuun ja tällöin XSLT-prosessorin on valittava instantioitava malli tiettyjen sääntöjen perusteella.

`xsl:include`-elementillä voidaan koostaa loogisesti yksi XSLT-dokumentti usean fyysisen tiedoston sisällöstä [22]. Mallien prioriteettien kannalta tilanne on tällöin sama kuin että kaikki mallit olisivat alun perinkin olleet samassa tiedostossa.

`xsl:import`-elementillä voidaan sisällyttää muiden XSLT-dokumenttien malleja, mutta tällöin prioriteetteihin liittyy oma käsittelynsä; *import-presedence* [22]. Tässä kutsujalla, eli `xsl:import`-elementin sisältävän XSLT-dokumentin malleilla on aina suurempi prioriteetti sisällytettyihin nähden. Sisällytetyistä malleista rakentuu prioriteettien mukaisesti puumainen hierarkia, jossa pienemmällä prioriteetilla olevia malleja voidaan kutsua erillisellä `xsl:apply-imports`-elementillä.

Kätevä keino välttää konflikteja on määritellä malleille moodeja (mode) [22]. Tällöin kutsuja määrittää `xsl:apply-templates`-kutsussa käytettäväksi tietyn moodin `mode`-attribuutilla ja kutsuttavan mallin on määritettävä sama moodi `xsl:template`-elementin vastaavannimisessä attribuutissa. Lisäksi `priority`-attribuutilla voidaan suoraan kokonaislukuna määrittää erilaiset prioriteetit esimerkiksi kahdelle muuten samanlaiselle mallille. Tätä voidaan käyttää hyväksi esimerkiksi kehitysvaiheessa estämään tiettyjen mallien instantiointi.

Kun import-hierarkia, moodit ja eksplisiittiset prioriteetit eivät vielä auta ratkaisemaan konfliktia, siirtyy XSLT-prosessori tarkastelemaan hahmojen eroja. Tässä pätevät seuraavat säännöt [41]:

- jos hahmossa käytetään unionia |, sen eri osilla on sama prioriteetti
- hahmo, jossa on askeleita ja siis hierarkista informaatiota, on ensisijainen yksiaskeleiseen nähden
- elementin tai attribuutin nimi on ensisijainen suhteessa ”villeihin kortteihin”, kuten *
- hahmo, jossa on mukana predikaatti (ehto hakasuluissa), on ensisijainen predikaatittomaan nähden
- jos yllä olevat ehdot eivät riitä, *voidaan* käyttää esimerkiksi mallien järjestystä XSLT-dokumentissa hyväksi

Seuraavassa esimerkissä olevista malleista valittaisiin HTML-dokumentin `title`-elementin käsittelyyn ensimmäinen versio, sillä sen `match`-attribuutissa oleva hahmo on rakenteellisesti tarkempi:

```
<xsl:template match="/html/head/title">
```

```
  Suurempi prioriteetti.
</xsl:template>

<xsl:template match="title">
  Pienempi prioriteetti.
</xsl:template>
```

3.3.6 Tulostus

XSLT-muunnoksen tulospuun tulostusta merkkijonomuotoon voidaan säädellä `xsl:output`-elementillä [16]. Elementillä voidaan mm. valita tulostustapa (xml, html, text) ja merkkeihin sovellettava koodaus (UTF-8, ISO-8859-1 jne.). Tässä työssä käytetään XML-tulostustilaa, joten ainoastaan sitä käsitellään tarkemmin. XML-tulostustilassa tulospuusta tehdään joko hyvinmuotoiltu XML-dokumentti tai mikäli juurella on useita lapsisolmuja, hyvinmuotoiltu ulkoinen yleinen jäsennetty entiteetti (external general parsed entity) [16]. Jälkimmäinenkin on käytännössä XML-dokumentti, jossa rikotaan ainoastaan sääntöä, että dokumentilla saisi olla vain yksi juurisolmu.

Perustoiminnot lähdepuun sisällön siirtämiseksi tulospuuhun ovat `xsl:value-of`, `xsl:copy` ja `xsl:copy-of` [16]. `xsl:value-of`-elementillä voidaan tulostaa minkä tahansa XPath-lausekkeen tuloksena syntyvä sisältö tulospuuhun. Ennen tulostusta sisältö muunnetaan merkkijonoksi samoin kuin se tapahtuisi `string()`-funktion avulla. `xsl:copy-of` kopioi rekursiivisesti kontekstisolmun ja sen lapset. `xsl:copy` kopioi kontekstisolmun ja sen nimiavaruuden, muttei mitään muuta. `xsl:for-each`-elementillä voidaa iteroida jonkin valitun solmujoukon yli.

```
<!-- kopioidaan koko dokumentti -->
<xsl:copy-of select="/" />

<!-- iteroidaan HTML-dokumentin juuritason kappaleet ja
     tehdään niistä lihavoituja -->
<xsl:for-each select="/html/body/p">
  <p><b>
    <xsl:value-of select="." />   <!-- tämä valitsee vain sisällön -->
  </b></p>
</xsl:for-each>
```

Literaali tuloselementti (literal result element) on mikä tahansa XML-elementti, joka on esitetty mallissa sellaisenaan, ei kuulu XSLT-nimiavaruuteen ja kirjoitetaan sellaisenaan tulospuuhun [22]. Tällaisten elementtien pitää olla hyvinmuotoiltua XML-merkkijonotietoa.

Literaalit tuloselementit eivät ole ainoa tapa tulostaa sisältöä tulospuuhun vaan voidaan käyttää myös XSLT-nimiavaruudessa olevia XSLT-käskyjä (konstruktoreja), kuten `xsl:element` [22]. Vastaavat konstruktorit on olemassa myös attribuuteille (`xsl:attribute`), kommenteille (`xsl:comment`) ja prosessointiohjeille (`xsl:prosessing-instruction`). Alla on esimerkki, jossa tulostetaan kaksi elementtiä; ensimmäinen käyttäen `xsl:element`-konstruktoria ja toinen käyttäen literaalia elementtiä:

```
<xsl:template name="foo">
  <xsl:element name="esimerkki">1</xsl:element>
  <esimerkki>2</esimerkki>
</xsl:template>
```

Edellinen esimerkki tuottaisi seuraavanlaista XML-sisältöä:

```
<esimerkki>1</esimerkki>
<esimerkki>2</esimerkki>
```

3.3.7 Hyödyllisiä XSLT-elementtejä ja -funktioita

XSLT-tekniikassa on käytettävissä ehdollisten tilanteiden hallintaan kaksi muistakin ohjelmointikielistä tuttua rakennetta [22]. `xsl:if`-elementillä on yksi attribuutti, `test`, joka sisältää Boolen-arvoisen lausekkeen. Mikäli tämä arvo on tosi, suoritetaan elementin sisältö. `xsl:choose`-elementillä voidaan valita useammasta vaihtoehdosta (`xsl:when`), joista kukin sisältää ehdon `test`-attribuuttina.

XPath 1.0 määrittelee 27 funktiota, joita voidaan kutsua lausekkeissa, kuten XSLT:ssä käytettyjen hahmojen predikaateissa [22]. XSLT-suositus lisää valikoimaan 9 uutta funktiota. XSLT-funktioilla voidaan esimerkiksi ladata useampia XML-dokumentteja käsiteltäväksi (`document()`), luoda muunnosta tehostavia hajautusrakenteita (`key()`), muotoilla numeroarvoja (`format-number()`) ja tutkia mahdollisten toteutuskohtaisten laajennusfunktioiden ja –elementtien käytettävyyttä (`function-available()`, `element-available()`).

Aina mallien tulospuuhun kirjoittamat solmut eivät ole aivan halutussa järjestyksessä. Tällöin voidaan käyttää XSLT:n `xsl:sort`-elementtiä ohjaamaan tulostusta aakkoselliseen tai numeeriseen järjestykseen [22].

3.3.8 Whitespace-käsittely

XSLT-prosessori noudattaa tiettyjä, tarkkoja sääntöjä lukiessaan ns. whitespace-merkkejä XSLT- ja XML-dokumenteista. Seuraavat säännöt pätevät aina [31], jollei whitespace-käsittelyä erikseen ohjata XSLT:n keinoin:

- tekstisolmu otetaan aina mukaan, mikäli se ei koostu pelkistä whitespace-merkeistä (#x20, #x9, #xD, #xA)
- jos tekstisolmun esi-isällä on `xml:space`-attribuutti arvolla `preserve`, eikä puussa lähemmällä esi-isällä ole vastaavaa attribuuttia arvolla `default`, whitespace-merkkejä sisältävät tekstisolmutkin otetaan mukaan
- ainoa XSLT-dokumentin elementti, jonka sisältämät pelkkiä whitespace-merkkejä sisältävät tekstisolmut otetaan mukaan ilman eri määrittelyjä, on `xsl:text`

3.3.9 Prosessointimalleista

Yleisimmin kuvattu prosessointitapa tunnetaan nimellä *push*-malli tai *document-driven transformation*, koska prosessointijärjestystä ohjaa lähdepuuna kuvattu dokumentti, eikä XSLT-kieleen perustuva ”ohjelma” [54]. Käsittely perustuu suoraan lähdedokumentin sisältöön ja elementtien järjestykseen siinä. Tässä mallissa käytetään mallien instantiointiin `xsl:apply-templates` –elementtiä,

mallit ovat nimettömiä ja sopiva malli valitaan eri kandidaattien `match`-attribuutissa olevan XPath-hahmon perusteella.

Push-malli ei ole paras mahdollinen kaikkiin tilanteisiin, mutta erityisesti jos lähdedokumentin elementtien järjestyksellä on merkitystä lopputuloksen kannalta, on siitä etua alla kuvattavaan *pull*-malliin nähden [21].

Pull-mallissa (stylesheet-driven transformation) XSLT-dokumentti ohjaa käsittelyä ja tietoa otetaan lähdedokumentista tarpeen mukaan [54]. Mallien kutsut tehdään `xsl:call-template` –elementillä ja valittava malli on siten täysin kutsujan määrättävissä. Lähdedokumentista haetaan sisältöä suoraan `xsl:value-of`-elementtien avulla.

3.3.10 Ilmaisuvoima, deklaratiivisuus ja funktionaalisuus

XSLT-kieli on ilmaisuvoimaltaan samalla tasolla kuin *oikeat* ohjelmointikielet. Kepser osoittaa sen olevan *Turing-täydellinen* kuvaamalla ns. μ-rekursiiviset funktiot XSLT-kielen avulla [27]. Samassa paperissa todistetaan myös XML Query –kielen (XQuery) Turing-täydellisyys.

XSLT:a on pidetty deklaratiivisena ohjelmointikielenä, sillä muuttujille voidaan antaa arvoja, mutta niitä ei voida jälkeenpäin muuttaa. Artikkelissaan [36] Novachev osoittaa esimerkein, että XSLT:lla voidaan toteuttaa käsittely, joka muistuttaa funktionaalista ohjelmointia. XSLT:n mallit eivät ole *ensimmäisen luokan kansalaisia*, kuten funktionaalisten kielten funktiot. Niitä ei siis voida välittää suoraan toisille malleille (funktioille) parametreina. Silti voidaan toteuttaa vastaava mekanismi, jossa tieto valittavasta mallista välittyy käsiteltävien elementtien uniikkien nimiavaruuksien avulla.

3.3.11 XSLT:n laajentaminen

XSLT-kieltä voidaan laajentaa kahdella tavalla; funktioilla ja elementeillä [16]. XSLT 1.0 ei sisällä määrittelyä mekanismista näiden laajennusten toteuttamiseen ja käytettävän XSLT-prosessorin ei myöskään voida olettaa tuntevan juuri haluttuja laajennuksia. Tätä varten XSLT sisältää funktiot laajennusten käytettävyyden tarkistamiseen (`element-available()`, `function-available()`). Mikäli tätä tarkistusta ei tehdä jollekin laajennuselementille ja se ei ole käytettävissä, voidaan tilanne käsitellä hallitusti `xsl:fallback`-elementin avulla. Esimerkki:

```
<xsl:template match="*">
  <xsl:choose>
    <xsl:when test="function-available('oma:laajennusfunktio')">
      Funktio löytyi.
    </xsl:when>
    <xsl:otherwise>Funktiota ei ollut.</xsl:otherwise>
  </xsl:choose>
  <oma:laajennus select="*">
    <xsl:fallback>
      <xsl:message terminate="yes">
        oma:laajennus ei ollut käytettävissä!
      </xsl:message>
    </xsl:fallback>
  </oma:laajennus>
```

```
</xsl:template>
```

3.3.12 EXSLT-laajennukset

EXSLT [47] on XSLT-kehittäjien yhteinen yritys koota yhteen ja standardoida XSLT-laajennuksia. Laajennukset on jaettu useaan eri moduuliin käyttötarkoituksen mukaan. XSLT-prosessorien toteuttajia pyritään rohkaisemaan laajennusten toteuttamiseen näiden määritysten mukaisesti. Näin saadaan parannettua XSLT-dokumenttien siirrettävyyttä. Suurin osa EXSLT-laajennuksista on funktioita. XSLT-prosessorien on siis suoraan tuettava niitä. Monet laajennukset on kuitenkin toteutettu myös erillisinä XSLT-malleina, jotka toimivat missä tahansa XSLT 1.0-prosessorissa.

EXSLT ei ole ainoa yritys standardoida XSLT-laajennuksia. Käytännössä se on kuitenkin XSLT-kehittäjien yhteisön todennäköisesti parhaiten tukema ja laajasti viitattu.

3.4 XSL-FO

Extensible Stylesheet Language – Formatting Objects on muotoilukieli, jolla voidaan määritellä tarkka ulkoasu esimerkiksi tulostettaville dokumenteille [40]. XSL-FO on verrattavissa PostScript tai Portable Document Format (PDF) – kieliin. XSL-FO:ta voidaankin käyttää välimuotona, kun halutaan tuottaa PDF-dokumentteja. XSL-FO-kieltä ei tässä työssä hyödynnetä.

3.5 Huomioita ja kritiikkiä

XSLT 1.0 –suositus toteaa, että XSLT ei ole tarkoitettu käytettäväksi täysin yleisenä XML-muunnostekniikkana vaan erityisesti suunniteltu sellaisia muunnoksia varten, mitä tarvitaan, kun käytetään myös XSL-FO-kieltä [16]. Tämän lähtökohdan vuoksi ei XSLT:ltä voidakaan vaatia täydellistä yleisyyttä.

Sal Manganon *XSLT CookBook*:ssa esitetään laaja kirjo erilaisia laajennustapoja tilanteisiin, mitä ei selvästikään ole ajateltu XSLT:llä ratkaistavaksi. Osa toteutuksista on hyvin monimutkaisia ja olisivat varsin helppoja tehdä tyypillisillä imperatiivisilla tai olio-ohjelmointikielillä. Tästä ja joiltakin erilaisia XSLT-*niksejä* sisältäviltä WWW-sivuilta jää sellainen maku, että ratkaisut olisi ehkä ollut syytä tehdä jollakin toisella tekniikalla, tai XSLT:tä itseään kehitettävä.

XSLT-suosituksen puuttellisuus laajennusten toteutustavan suhteen tuo ongelmia XSLT-dokumenttien siirrettävyyteen eri muunnosprosessoreilla suoritettavaksi. XSLT 2.0 ei korjaa tätä ongelmaa, mitä onkin kritisoitu [31].

3.6 W3C:n uudet suositukset

W3C:lla on tätä kirjoitettaessa standardointivaiheessa joukko uusia suosituksia. Näiden myötä XSLT-tekniikka tulee saamaan rinnalleen uuden kyselyihin painottuvan XML Query 1.0 –kielen. XSLT- ja XPath-tekniikoista julkaistaan samalla uudet versiot; XSLT 2.0 ja XPath 2.0. Sekä uusi XML Query että XSLT tukeutuvat samaan XPath-versioon ja tietomalliin.

XPath 2.0:n uusi tietomalli tuo mukaan XML Schema-suosituksen tietotyypit ja mahdollisuuden määritellä uusia tietotyyppejä. Myös funktioiden parametrit on mahdollista tyypittää. [33]. Järjestämättömistä solmujoukoista on luovuttu ja tilalla on aina järjestetty sekvenssi (sequence). Käytettävissä olevien funktioiden määrä on huomattavasti laajentunut ja sisältää uutena esimerkiksi säännöllisten lausekkeiden käsittelyn.

XSLT 2.0 tarjoaa lukemattoman joukon muutoksia sen lisäksi, että se tukeutuu uuteen XPath-versioon. Mallien tuloksia voidaan käsitellä suoraan sekvensseinä ja siten soveltaa niihin XSLT ja XPath-toiminnallisuutta, mikä oli aiemmin tehtävä laajennusten avulla [22]. Tulospuita voi olla useampia ja tulostuksessa tuetaan XML:n lisäksi erityisesti XHTML-kieltä. Voidaan käyttää säännöllisiä lausekkeita ja luoda omia XSLT-dokumenttikohtaisia täysin siirrettäviä (portable) laajennusfunktioita.

XQuery (1.0) on XML-dokumenttien käsittelyyn tarkoitettu funktionaalinen ohjelmointikieli. XQuery käyttää osanaan XPath-kieltä. XQueryä on kehitetty jo sangen pitkään, eikä standardia ollut vielä julkaistu tämän työn alkumetreillä, joten tekniikka hylättiin tarkasteltavana vaihtoehtona XSLT:lle. XQueryä voitaisiin toki käyttää käyttöliittymän tuottamisessa, mitä esimerkiksi Bothner artikkelissaan [10] hyvin käytännönläheisesti valaisee.

4. Luku - Java-tekniikat

Lopullisen toteutuksen toimintaympäristönä on J2EE-määrittelyn (Java 2 Enterprise Edition) mukainen palvelinalusta, joten siitä hyödynnettäviä tekniikoita kuvataan lyhyesti. Erityisesti kuvataan JSP- ja JSTL-tekniikat, joita käytetään käyttöliittymäsivujen tuottamiseen dynaamisesti XML-dokumenteista.

4.1 Java 2 Enterprise Edition

Java 2 Enterprise Edition (J2EE) on standardi [44], joka määrittelee Java-ohjelmointikielen päälle rakentuvan monikerrosarkkitehtuurin. J2EE-standardin määrittely tapahtuu Java Community Process –organisaation alaisuudessa ja siihen osallistuu useita eri ohjelmistoyrityksiä ja yksityishenkilöitä. Uusin määritys, J2EE 1.4, on marraskuulta 2003 [45].

J2EE-määritys kuvaa joukon rajapintoja ja konfiguraatioita. J2EE-sovelluspalvelinohjelmisto toteuttaa nämä rajapinnat ja tulkitsee rajapintoja käyttävien sovellusten konfiguraatiotiedostoja. Markkinoilla on lukemattomia kaupallisia ja ilmaisia J2EE-sovelluspalvelimia.

J2EE-standardin määritykset voidaan jakaa kolmeen osaan; komponenttitekniikat, palvelut ja kommunikaatio [48]. Komponenttitekniikat mahdollistavat sovellusten käyttöliittymän ja liiketoimintalogiikan toteuttamisen uudelleenkäytettävinä kokonaisuuksina. Komponentteja tuetaan J2EE-järjestelmäpalveluilla, jotka helpottavat sovelluskehitystä ja mahdollistavat sovellusten konfiguroinnin sen mukaan, mitä järjestelmäpalveluita milläkin J2EE-sovelluspalvelimella on saatavilla. Järjestelmäpalveluita ovat esimerkiksi tietokantayhteydet, yhteydet muihin yrityksen keskitettyihin tietojärjestelmiin (EIS, Enterprise Information System), tapahtumanhallinta, nimi- ja hakemistopalvelut ja asynkroninen kommunikointi. Kolmanteen kategoriaan kuuluu kommunikointi J2EE-asiakasohjelmien ja komponenttien kesken näiden fyysisestä sijainnista riippumattomasti.

Tässä työssä käytetään J2EE 1.3 –määrityksiin kuuluvia Servlet 2.3 ja JavaServer Pages 1.2 –tekniikoita.

4.2 Java API for XML Processing

Java API for XML Processing (JAXP) [49] on kokoelma rajapintoja XML-käsittelyyn Java-ympäristössä. JAXP pitää sisällään sekä SAX että DOM –mallien mukaiset jäsennysrajapinnat. Lisäksi JAXP tukee XSLT-muunnoksia ja XML-nimiavaruuksia. Rajapintojen *takana* voidaan käyttää haluttuja jäsennin- ja muunnintoteutuksia, jotka valitaan käytettäviksi joko konfiguraatiotiedostojen avulla tai ohjelmallisesti. Näin siis Java-ohjelmissa ja vaikkapa JSP-sivuilla voidaan JAXP-rajapintoja hyödyntäen käsitellä XML-dokumentteja SAX, DOM ja XSLT-tekniikoilla.

Java 2 Standard Edition Software Development Kit (J2SE SDK) sisältää oletusarvoisesti Apache-projektissa kehitetyn Crimson-parserin ja Xalan-Java 2 – XSLT-prosessorin. Tässä työssä jäsentämiseen käytettävä Xerces2 [4] on JAXP-rajapinnan mukainen jäsennin ja se toteuttaa sekä DOM että SAX-mallin mukaiset jäsennysrajapinnat. Xerces2 on myös uuden, vielä *beta*-asteella olevan J2SE SDK 1.5:n oletusjäsennin.

4.3 JavaServer Pages

4.3.1 Yleistä

JavaServer Pages (JSP) on J2EE-teknologia, jonka avulla voidaan tuottaa dynaamisia käyttöliittymiä [38]. JSP-tekniikka rakentuu Servlet-tekniikan päälle. Servletit ovat Java-luokkia, jotka laajentavat J2EE-sovelluspalvelimen Web-palvelinosuuden toiminnallisuutta [23]. Servlettien ajoympäristö (servlet container) huolehtii Servlettien alustamisesta ja kutsujen välittämisestä niille.

JSP-sivu on aluksi olemassa lähdekoodina. Useimmiten se on sekoitus HTML-koodia, Java-ohjelmointikieltä, JSP-direktiivejä ja –toimintoja, jotka määrittävät JSP-sivulla luotavan HTML-sivun sisällön dynaamisesti [23]. Kaikki JSP-sivulla oleva sisältö, mikä ei ole jotain alla kuvattavista sallituista toiminnallisista rakenteista, kirjoitetaan suoraan tulostuspuskuriin. JSP-sivu voi myös olla täysin tyhjä tai sisältää vain staattista tietoa.

JSP-sivu muunnetaan Java-ohjelmakoodiksi, joka toteuttaa HttpServlet-rajapinnan [23]. Tämä Java-luokka käännetään ja käännöksen lopputuloksena syntyy varsinainen ajonaikainen versio JSP-sivun toteutuksesta. Näin JSP-sivuilla on käytettävissä kaikki Servlettien toiminnallisuus ja lisäksi joitakin JSP-tekniikalle spesifejä piirteitä. JSP-sivuista voidaan ajonaikaisesti asentaa uusia versioita, sillä J2EE-sovelluspalvelimen JSP-palvelin (JSP container) kääntää ja ottaa uudet versiot käyttöön automaattisesti.

4.3.2 Direktiivit

Direktiivit ovat ohjeita JSP-palvelimelle ja ohjaavat deklaratiivisesti sitä, millaista Java-ohjelmakoodia JSP-sivusta generoidaan [23]. JSP 1.2 –määritys kuvaa kolme direktiiviä; `page`, `include` ja `taglib`. Direktiivien syntaksi noudattaa seuraavaa kaavaa, missä attribuuttit ovat direktiivikohtaisia:

```
<%@direktiivin_nimi attribuutti1="arvo" attribuutti2="arvo" %>
```

page-direktiivillä määritellään koko sivua koskevia attribuutteja, joita on lukuisa joukko. contentType–attribuutti määrittää selaimelle lähetettävän sisällön tyypin ja merkistön. import määrittää sivusta luotavaan Java-ohjelmakoodiin tehtävät vastaavat Java-luokkiin viittaavat import-määritykset. session määrittää, vaatiiko JSP-sivun prosessointi http-istunnon luomista sovelluspalvelimeen, jos sellaista ei muuten ole vielä olemassa. isThreadSafe määrittää, voidaanko generoitua Java-luokkaa kutsua useasta säikeestä rinnakkain Muilla page-direktiivin attribuuteilla voidaan vaikuttaa esimerkiksi tulostuspuskurin kokoon ja käyttäytymiseen, sivun erityiskäsittelyyn virhesivuna ja sivun lähdekooditiedoston merkkikoodaukseen.

include-direktiivi sisällyttää JSP-sivuun toisen resurssin sisällön ennen käännöstä:

```
<%@include file="tiedoston_nimi" %>
```

taglib-direktiivillä otetaan käyttöön laajennustoimintoja (tageja) määrittämällä käytettävä tagikirjasto URI-viitteellä ja sivulla käytettävä etuliite ko. tagikirjaston tageille. Tagikirjastoja kuvataan alla laajemmin.

```
<%taglib uri="tagikirjaston URI" prefix="sivulla käytettävä etuliite" %>
```

4.3.3 Skriptaus-elementit

Skriptaus-elementtejä (scripting element) ovat Java-kieliset lausekkeet (expression), skriptletit (scriptlet) ja esittelyt (declaration).

Lausekkeilla voidaan tehdä mitä tahansa Java-kutsuja ja tulostaa kutsun tuottama merkkijono suoraan tulostuspuskuriin [23]. Ehtona tälle on ainoastaan, että lausekkeen arvo on muunnettavissa merkkijonomuotoon. Skriptletit ovat Java-ohjelmakoodia, joka liitetään JSP-sivusta tuotettavaan Servlet-lähdekoodiin sellaisenaan [23]. Deklaraatioiden sisältämä Java-koodi sisällytetään generoitavaan Java-luokkaan kaikkien metodien ulkopuolelle. Deklaraatioilla voidaan siten määritellä esimerkiksi kokonaisia metodeja tai luokan kaikille metodeille näkyviä instanssimuuttujia:

```
<%= Java-lauseke %>

<% Java-lauseita %>

<%! muuttujien ja metodien esittelyjä %>
```

4.3.4 Toiminnot, tagit

Toiminnot (action) ovat korkeamman tason JSP-elementtejä [23]. Ne kirjoitetaan aina XML-syntaksia noudattaen, toisin kuin muut JSP-sivun toiminnalliset elementit:

```
<tagin_nimi [attribuutti="arvo"]* > sisältöä </tagin_nimi>
```

JSP 1.2-standardi määrittää seitsemän toimintoa [23]. `jsp:useBean` määrittää sivulla käytettäviä tietoja sisältävän JavaBean-ilmentymän ja liittää sen johonkin muuttujaan. `jsp:setProperty` ja `jsp:getProperty` –toiminnoilla voidaan käsitellä JavaBeanin tietoja. `jsp:include` kutsuu ajonaikaisesti toista JSP-sivua ja kirjoittaa sen tulostuksen tämän toiminnon kohdalle tulostuspuskuriin. `jsp:forward` siirtää käsittelyn kokonaan toiselle JSP-sivulle tyhjentäen tulostuspuskurin, mikäli sinne on jo kirjoitettu sisältöä. `jsp:param` määrittää include tai forward-toiminnolla kutsuttavalle resurssille välitettäviä parametreja. `jsp:plugin` tuottaa HTML-elementin, jonka avulla selain voi ladata Java-laajennuksen ajaakseen esimerkiksi Applet-koodia.

Laajennustoiminnot (custom action, tag extension, custom tag) ovat syntaksiltaan seitsemän perustoiminnon kaltaisia, mutta kehittäjän itse määrittämiä laajennuksia. Näitä kuvataan laajemmin kohdassa *Tagikirjastot*.

4.3.5 Kommentit

JSP-sivulle voidaan sisällyttää kommentteja, joiden sisältöä ei millään tavalla käsitellä eikä tulosteta [23]:

```
<%-- kommentti --%>
```

4.3.6 Implisiittiset oliot

Implisiittiset oliot ovat Java-olioita, jotka määrittävät toimintaympäristön JSP-ohjelmalle. JSP-palvelin antaa ne jokaisen JSP-sivun käyttöön ja niihin voidaan viitata suoraan skriptleteissä ja lausekkeissa [23]. Tärkeimmät neljä implisiittistä oliota liittyvät suoraan JSP-sivun ajoympäristön eri konteksteihin:

- `page` on viite JSP-sivusta generoidun Java-luokan ilmentymään itseensä. Tähän kontekstiin voidaan asettaa juuri kyseisen JSP-sivun prosessoinnissa tarvittavia attribuutteja.
- `request` on `HttpServletRequest`-olio, joka kuvaa palvelimen vastaanottamaa http-kutsua ja on voimassa kutsun käsittelyn ajan.
- `session` on `HttpSession`-olio, mikäli JSP-sivun käsittelyyn liittyy istunto. Olio on sama koko istunnon voimassaolon ajan.
- `application` on `ServletContext`-olio, johon sisältyy koko Web-sovellukselle yhteiset parametrit. Olio on yhteinen kaikille istunnoille.

Muista implisiittisistä olioista mainittakoon tulostuspuskuria edustava `JspWriter`-ilmentymä `out`, virhesivuilla käytettävissä oleva poikkeustilanteen kuvaava `exception` sekä `pageContext`, joka on vaihtoehtoinen tapa viitata yllä kuvattuihin neljään kontekstiin.

4.3.7 Muita ominaisuuksia

J2EE-web-sovelluksiin liittyy runsaasti ominaisuuksia, joita ei tässä työssä millään tavoin tarvita. Näitä ovat esimerkiksi istunnonhallinta, sovelluksen

tapahtumankuuntelijat ja suodattimet, joita voidaan asettaa sovelluksen *eteen* suorittamaan kaikille kutsuille tai vastauksille yhteistä käsittelyä.

4.4 Tagi-laajennukset

4.4.1 Yleistä

JSP-sivujen ominaisuuksia voidaan laajentaa omilla XML-syntaksia noudattavilla toiminnoilla eli tageilla [23]. Tagit paketoidaan tagikirjastoihin (custom tag library, tag extension library). Tagikirjastojen avulla voidaan esimerkiksi erottaa liiketoimintalogiikka kehitettäväksi erillään käyttöliittymästä. Käyttöliittymän ohjelmoinnista tulee yksinkertaisempaa ja koodista selkeämpää. Tagikirjastojen sisältämää toiminnallisuutta voidaan uudelleenkäyttää. Kehitysvälineet (IDE, Integrated Development Environment) voivat hyödyntää tagikirjastoja esimerkiksi näyttämällä kehittäjille tietoa käytetyistä tageista ja validoimalla tageille välitettävät attribuutit kehitysaikaisesti.

4.4.2 Tagikirjaston osat

Tagikirjaston sisältämien tagien toiminta kuvataan XML-dokumentissa, jota kutsutaan nimellä *Tag Library Descriptor (TLD)* [23]. TLD määrittää tagien nimet ja attribuutit ja liittää niihin varsinaisen toiminnallisuuden toteuttavat Java-luokat. TLD-tiedosto sisällytetään yleensä Web-sovelluksen sisälle hakemistoon `/WEB-INF/tlds` tai se voi olla paketoituna tagikirjaston mukaan.

Tagikirjasto identifioidaan yksikäsitteisellä XML-nimiavaruuden kuvaavalla URI-viitteellä [23][8]. Tagikirjastoa käyttävän Web-sovelluksen `web.xml`-kuvailutiedostoon määritellään kuvaus tämän URI:n ja tagikirjaston TLD-tiedoston välille. Tagikirjastoa käyttävällä JSP-sivulla viitataan tagikirjaston URI:in tai suoraan TLD-tiedostoon absoluuttisella polulla:

```
<%@taglib  prefix="suora_viite" uri="/WEB-INF/tlds/taglib.tld" %>
<%@taglib  prefix="web_xml_kuvattu_viite"
           uri="http://yksikasitteinen_uri" %>
```

Tagien ajonaikainen toiminnallisuus ohjelmoidaan Java-luokkiin, jotka toteuttavat jonkin rajapinnoista `Tag`, `IterationTag` tai `BodyTag` tai laajentavat luokkaa `TagSupport` tai `BodyTagSupport` [23]. Laajennettavat luokat ovat olemassa toteutustyön helpottamiseksi; tagikirjaston ohjelmoijan ei tarvitse kirjoittaa sellaisia rajapinnoissa olevia metodeja, joiden osalta pärjätään perus- tai tyhjillä toteutuksilla. Tagikirjasto paketoidaan tyypillisesti yhteen JAR-tiedostoon (Java Archive).

Lisäksi tagikirjastolla voi olla käännösaikaista käsittelyä varten yleinen validointiluokka `TagLibraryValidator` ja tagikohtaiset `TagExtraInfo`-validointiluokat. Tagikohtaiset validointiluokat ovat vanhentuneita ja kirjastokohtaisen validointiluokan käyttöä suositellaan [38].

4.4.3 Prosessointimalli

Aluksi JSP-palvelin jäsentää JSP-sivun [38]. Tässä vaiheessa huomioidaan sivun `include`-direktiivit koostamalla sisällytetyistä sivuista yksi kokonaisuus ja tulkitaan sivuilla käytettävien tagikirjastojen TLD-määrittelyt.

Validointivaiheessa tarkistetaan tagikirjastojen kutsut [38]. Jokaiselle kirjastolle etsitään validointiluokkaa ja jos sellainen löytyy, sivun sisältö annetaan ko. luokalle tarkistettavaksi. Jokaiselle tagille etsitään erillistä `TagExtraInfo`-luokkaa, jolla voidaan tehdä tagikohtaista käsittelyä.

Validoitu JSP-sivu muunnetaan Java-lähdekoodiksi. Tagikirjastojen kutsut muunnetaan Java-luokkien kutsuiksi ja koodiin lisätään tagien mahdollisesti tarvitsemat muuttujat (scripting variable). Lopuksi lähdekoodi käännetään tavukoodiksi ja tallennetaan class-tiedostoksi.

JSP-sivu ajetaan käännettynä Java-Servlettinä. Tagikirjastojen käyttö tapahtuu tagit toteuttavien Java-luokkien ilmentymien avulla.

4.4.4 Tagien käyttö

Tagikirjasto otetaan käyttöön JSP-sivulla yllä kuvatun `taglib`-direktiivin avulla. Tämä liittää tagikirjastoon halutun etuliitteen, jolla tagikirjaston toteutuksiin viitataan JSP-sivulla [38]. Tageille voidaan välittää parametreja käyttämällä tagikohtaisesti määriteltyjä XML-attribuutteja. Alla on kokonainen esimerkki JSTL-tagikirjaston käytöstä, jossa ensin asetetaan String-tyyppinen attribuutti JSP-sivun kontekstiin ja sen jälkeen tulostetaan attribuutin arvo suoraan tulostuspuskuriin:

```
<%@taglib uri="http://java.sun.com/jstl/ea/core" prefix="c" %>
<c:set var="attribuutti" scope="page" value="Hello World!" />
<c:out value="${pageScope.attribuutti}" />
```

4.4.5 Tulostuksen puskurointi

Tagien sisällön tulostukseen liittyy erityinen `BodyContent`-luokka [23]. Siitä välitetään haluttaessa oma ilmentymänsä kullekin tagille yleisen tulostuspuskurin sijaan. Näin tietty tagi voi puskuroida sisältämistään JSP-elementeistä syntyvän tulostuksen ja joko jatkokäsitellä sitä tai kirjoittaa sen suoraan osaksi lopullista JSP-sivun tulostusta. Tietyllä JSP-tagilla on siis täysi kontrolli siihen, mitä hierarkiassa sen sisällä olevat tagit voivat tulostaa.

4.5 JSP Standard Tag Library

JSP Standard Tag Library (JSTL) on joukko tagikirjastoja. Sitä kehitettiin alun perin Apache-projektissa mutta on nyt standardoitu Java Community Process-yhteisössä (JCP) standardinumerolla JSR-052 [23][19]. JSTL tuli mukaan J2EE-määrittelyihin versiossa 1.4 ja on tarkemmin osa siihen kuuluvaa JSP 2.0-tekniikkaa. Tässä työssä käytetään kuitenkin vielä JSTL 1.0 –versiota, joka tukeutuu JSP 1.2 –tekniikkaan.

JSTL koostuu neljästä osakirjastosta, joita kutakin kuvataan alla. Kustakin kirjastosta on kaksi eri versiota. Tässä keskitytään versioihin, joissa on attribuuttien arvojen laskennassa käytettävissä JSP Expression Language-kieli (EL).

4.5.1 Expression Language (EL)

EL on JSTL-kirjastoon sisältyvä yksinkertainen lausekekieli, jota voidaan käyttää JSTL-tagien attribuuttien arvojen laskemiseen [46]. EL-lauseke kirjoitetaan attribuutin arvoksi ja ympäröidään merkeillä ${ ja }. EL on käytettävissä kaikissa muissa JSTL-kirjaston tageissa, paitsi xml-kirjastossa, missä `select`-attribuutin arvo tulkitaan EL:n sijaan XPath-lausekkeena.

Yksinkertaisimmillaan EL:lla voidaan viitata eri konteksteihin tallennettuihin olioihin eli attribuutteihin. Jos kontekstia ei erikseen määritellä, haetaan attribuuttia konteksteista järjestyksessä `page`, `request`, `session`, `application`. Esimerkki:

```
<c:out value="${attribuutin_nimi}" />
<c:oot value="${sessionScope.attribuutin_nimi}" />
```

Attribuutin arvon haku voidaan kohdistaa 11 eri kontekstiin. Normaaleiden JSP-kontekstien (`pageScope`, `requestScope`, `sessionScope`, `applicationScope`, `pageContext`) lisäksi EL:lla voidaan viitata http-kutsun parametreihin, otsikkotietoihin (header), sovelluksen alustusparametreihin ja evästeisiin (cookie).

EL tarjoaa yksinkertaisen tavan viitata tietorakenteisiin [46]. Erilaisten kokoelmaluokkien kuten `Map`, `Table` tai `List` sisältöön voidaan viitata piste- tai hakasulkunotaatiolla. Sama onnistuu muillekin Java-luokille, olettaen että luokka toteuttaa haettavan tiedon nimeä vastaavan `get`-metodin. Alla olevat kaksi lauseketta tekevät täsmälleen saman:

```
<c:out value="${rakenne.avain}" />
<c:out value="${rakenne['avain']}" />
```

Pistenotaatiota voidaan käyttää vain jos haettavan tiedon avain noudattaa Java-muuttujien nimeämissääntöjä, eikä siten sisällä esimerkiksi pisteitä tai miinusmerkkejä.

EL sisältää joukon yleisiä vertailu-, aritmeettisia ja loogisia operaattoreita. Mainittakoon tässä erityisesti unaarinen operaattori `empty`, jolla voidaan testata, onko lausekkeen arvo `null` tai tyhjä merkkijono. EL, kuten skriptikielet yleensäkin, sisältää laajan joukon automaattisia tyyppikonversioita. Näitä ei tässä käsitellä, sillä työssä esiintyvät tapaukset eivät monimutkaisia konversioita vaadi.

4.5.2 JSTL: core

JSTL-kirjaston osa *core* sisältää toimintoja tietojen kirjoittamiseen tulostuspuskuriin, muuttujien asettamiseen ja poistamiseen kontekstista, virheiden käsittelyyn, konditionaalien ja iteraatioiden hallintaan, URL-osoitteiden

muodostamiseen ja ulkoisten tiedostojen sisällyttämiseen osaksi JSP-sivun tulostusta [46].

`c:out`-tagilla kirjoitetaan sisältöä tulostuspuskuriin. Viiden XML:n sisäänrakennetun entiteetin käsittely voidaan parametroida siten, että entiteetit kirjoitetaan joko sellaisenaan tai ne koodataan entiteettiviitteinä. `c:set`-tagi asettaa muuttujia eri konteksteihin. `c:catch`-tagilla voidaan ottaa kiinni ko. elementin sisällön heittämät poikkeukset. Kaikki Java-kielen yleisestä `Throwable`-luokasta periytetyt poikkeukset käsitellään.

Yksinkertaisia ehdollisia tilanteita voidaan hallita `c:if`-tagilla ja `c:choose` antaa mahdollisuuden valita yhden useasta vaihtoehdosta. `c:choose` sisältää lisäksi mahdollisuuden määrittää oletusvalinta, mikäli yksikään ehdoista ei ole tosi.

`c:forEach`-tagi suorittaa saman käsittelyn joko Javan kokoelmaluokkien sisällölle tai erikseen määriteltävälle joukolle kokonaislukuindeksejä.

```
<c:catch var="virhe">
  <c:set var="muuttuja" scope="page"> Hello World! </c:set>
  <c:out value="${pageScope.muuttuja}" escapeXml="false" />
</c:catch>
<c:if test="${not empty virhe}">
  Virhe: <c:out value="${virhe}" />
</c:if>

<c:forEach var="alkio" items="${taulukko}>
  Taulukossa on alkio <c:out value="${alkio}" />
</c:forEach>

<c:choose>
  <c:when test="${param.kayttaja == 'admin'}">
    Tervetuloa, pääkäyttäjä!
  </c:when>
  <c:otherwise>
    Tervetuloa, <c:out value="${param.kayttaja}"/>
  </c:otherwise>
</c:choose>
```

Muut kirjaston tagit ovat `c:url` URL-osoitteiden muodostamiseen, `c:redirect` kutsujen uudelleenohjaukseen, `c:forTokens` merkkijonon paloitteluun ja `c:import` tietojen hakemiseen ulkoisista lähteistä.

4.5.3 JSTL: xml

JSTL-kirjaston osa *xml* sisältää toimintoja XML-dokumenttien käsittelyyn. Erityisesti mainittakoon `x:parse` XML-dokumenttien jäsentämiseen ja `x:transform` kokonaisen XSLT-muunnoksen suorittamiseen osana JSP-sivun käsittelyä. Näitä kahta tagia ei kuitenkaa tässä työssä käytetä.

Kaikilla muilla xml-kirjaston tageilla on vastineensa core-kirjastossa. Nämä ovat `x:set`, `x:out`, `x:if`, `x:choose` ja `x:forEach`. Näissä on tärkeimpänä erona se, että core-kirjaston tageille tyypilliset attribuutit `value` ja `test` on korvattu xml-kirjastossa attribuutilla `select` [46]. `select`-attribuutin sisältö on XPath-kielinen lauseke, jolla voidaan viitata sekä JSP-sivun konteksteissa oleviin attribuutteihin että käsiteltävän XML-dokumentin sisältöön. Koska xml-kirjaston

tagien toiminta vastaa hyvin läheisesti core-kirjaston toimintaa, esitetään tässä vain yksinkertainen esimerkki:

```
<x:forEach select="$pageScope:xml_dokumentti/*/*">
  Juurielementillä on lapsena elementti <x:out select="name()" />
</x:forEach>
```

xml-kirjaston tagit asettavat JSP-sivun kontekstien attribuutteihin arvot JSTL-toteutukselle spesifissä muodossa [19]. Tällaiset muuttujat ovat kuitenkin käytettävissä saman JSTL-toteutuksen sisäisesti kaikilla xml-kirjaston tageilla.

4.5.4 JSTL: fmt

JSTL:n *fmt*-kirjasto sisältää koko joukon kansainvälistämiseen (internationalization, i18n) ja tietojen muotoiluun (formatting) liittyviä toimintoja. Näillä voidaan konfiguroida kieli, maa ja aikavyöhyke ja näihin tietoihin perustuen tulostaa lokalisoituja tekstejä, päivämääriä ja numeroita.

`fmt:setLocale`-toiminnolla voidaan asettaa kieli ja niin haluttaessa myös maakoodi. Asetusta käytetään muiden toimintojen apuna. JSTL voi hyödyntää myös selaimen lähettämää tietoa halutusta kielestä, jos sitä ei erikseen määritellä [46]. `fmt:setBundle`-toiminnolla otetaan käyttöön lokalisoituja merkkijonomuotoisia tietoja sisältävä tietojoukko. Tässä voidaan käyttää nimi-arvo-pareista koostuvia tekstitiedostoja tai Java-luokkia. `fmt:message`-toiminnolla tulostetaan yksittäisiä merkkijonoja asetetusta tietojoukosta tai haluttu joukko voidaan määrittää suoraan tämän toiminnon attribuuttina. `fmt:parseNumber` ja `fmt:formatNumber` –toiminnoilla on mahdollista jäsentää kokonais- ja desimaalilukuja ja muotoilla niitä haluttuun muotoon. Vastaavat toiminnot, `fmt:parseDate` ja `fmt:formatDate` on käytettävissä päivämäärille.

```
<fmt:setLocale value="fi" />
<fmt:setBundle basename="kokeilu" scope="request" />
<fmt:message key="otsikko_1" />

<c:set var="desimaali" value="12345,60" scope="page" />
<fmt:formatNumber var="numero" scope="page"
    value="${pageScope.desimaali}" pattern="#,###.00" />
```

Tähän kirjastoon sisältyy useita muitakin toimintoja, joita ei tässä käsitelty.

4.5.5 JSTL: sql

JSTL-kirjasto sisältää kuusi relaatiotietokantojen käsittelyyn liittyvää toimintoa. Näillä SQL-toiminnoilla voidaan hakuja ja päivityksiä ja asettaa transaktioille rajat. SQL-kirjastoa ei tässä työssä käytetä ja sen käyttö JSP-sivuilla olisi ristiriidassa käyttöliittymän ja liiketoimintalogiikan erottamisen periaatteen kanssa.

4.6 Yhteenveto JSP:n modularisointimahdollisuuksista

Yllä on tullut esiin kaikenkaikkiaan neljä erilaista tapaa modularisoida JSP-sivuja. `jsp:include`-toiminnolla voidaan sisällyttää JSP-sivujen ja muiden tiedostojen osia sivuihin ajonaikaisesti samasta web-sovelluksesta. JSTL-tagikirjaston

`c:import`-toiminnolla voidaan sisällyttää tiedostoja mistä tahansa. `@include`-direktiivillä voidaan sisällyttää muita JSP-sivuja ennen käännöstä. Neljäntenä tapana on vielä omien Java-luokkien käyttö joko tagikirjastojen kautta tai suoraan JSP-sivulla. Tällä tavoin voidaan JSP-sivulta käsin tehdä mitä tahansa mitä Java-kielellä on toteutettavissa.

4.7 JSP 2.0

JSP 2.0 julkaistiin syksyllä 2003 osana J2EE 1.4 –määrityksiä. Sen päämääränä on yhä helpompi WWW-sovellusten kehittäminen, ilman varsinaisten ohjelmointikielten osaamista [30]. Expression Language –kielellä kirjoitettuja lausekkeita voidaan käyttää JSP 2.0-standardin mukaisilla sivuilla missä tahansa – ei ainoastaan JSTL-tagien attribuutteina. EL-kielellä voidaan kutsua myös omia staattisia funktioita. Tagien tekemistä on helpotettu uudella yksinkertaisemmalla rajapinnalla ja lisäksi tageja voidaan toteuttaa suoraan JSP-syntaksia käyttäen, normaaleiden Java-luokkien sijaan. Standardissa on korostettu JSP-sivujen XML-syntaksia eli *JSP-dokumentteja*. Sitä noudattamalla JSP-sivuilla voidaan helposti tuottaa hyvinmuotoiltuja XML-dokumentteja.

5. Luku - Tekniikoiden soveltaminen

Tässä luvussa esitellään edellä kuvattujen tekniikoiden käytännön sovelluksia ja niihin liittyviä käytäntöjä sekä keskeisiä tehokkuuteen vaikuttavia seikkoja.

5.1 XML:n soveltamisesta

XML-kieltä käytetään nykyisin laajasti erilaisiin tarkoituksiin, mihin sitä ei alun perin tarkoitettu. Tässä listataan muutamia käyttötapoja Johnsonin mukaan: [25]

- sovelluksen konfigurointi (configuration pattern)
- sovelluksen sisäisten tietojen rakenteellinen tallennus (structured storage pattern)
- tiedonvälitys eri sovellusten, järjestelmien ja yritysten välillä (data transmission pattern)
- ei-XML-muotoisten tietojen välittäminen XML-rakenteen osana (envelope pattern)
- deklaratiivinen ohjelmointikieli (declarative language pattern)
- etäkutsurajapinnan ja kutsujen kuvaaminen (interface definition pattern)

Tässä työssä sovellettava XSLT-muunnostekniikka sijoittuu yllä olevista kategorioista deklaratiivisiin ohjelmointikieliin.

5.1.1 XHTML

Extensible HyperText Markup Language (XHTML) on tunnetuin XML-kielen sovellus. Se on HTML 4.01-kielen formulointi XML-muotoon [53]. XHTML-kieltä ei tässä kuvata, vaan lukijan oletetaan tuntevan sen perusteet.

5.2 Suorituskyvyn optimointi

5.2.1 Java-virtuaalikoneista

Java-ohjelmien ajoympäristönä toimii Java-virtuaalikone (JVM, Java Virtual Machine). Tässä työssä käytetään Sun Microsystemsin HotSpot-virtuaalikonetta. Virtuaalikoneiden ominaisuuksista tässä mielenkiintoisia ovat erityisesti laitteistoriippumattoman Java-tavukoodin optimointi ja kääntäminen laitteistosidonnaiseen muotoon sekä viittaamattomien olioiden käyttämän muistitilan vapauttaminen eli roskankeruu (garbage collection).

HotSpot-virtuaalikone toteuttaa muutamia erilaisia algoritmeja roskankeruuseen [52]. Tässä työssä tehtävien nopeustestien vuoksi kiinnostava on erityisesti *Incremental Low-Pause Garbage Collector*, joka pyrkii jakamaan roskankeruuseen kuluvan ajan tasaisesti ohjelman suorituksen yhteyteen sen sijaan että se tapahtuisi harvoin ja siten suurina erinä kerrallaan. Tasainen roskankeruu ei ole oletusarvoinen mekanismi.

Optimointitekniikat ovat toinen nopeustesteihin merkittävästi vaikuttava seikka. Just-In-Time (JIT) –optimointi tarkoittaa tavukoodin kääntämistä ajoympäristön natiiviksi konekieleksi juuri ennen sen varsinaista suorittamista. Tämä hidastaa ohjelman suoritusta alkuvaiheessa eikä ole optimointikyvyiltään yhtä tehokas kuin HotSpot-virtuaalikoneen käyttämä tekniikka [52]. HotSpot-virtuaalikone tulkkaa tavukoodin aluksi sellaisenaan. Se analysoi ohjelman suoritusta ajonaikaisesti ja tunnistaa kuormitetuimmat kohdat. Optimointi ja natiiville konekielelle kääntäminen keskittyvät näihin kohtiin ohjelmassa. Optimointitekniikat ovat erilaiset työasema- ja palvelinkäyttöön tarkoitetuissa versioissa virtuaalikoneesta.

5.2.2 Kääntäminen ja tulkkaus

Useimmiten käännetty ohjelma on tehokkaampi kuin tulkattu ja näin näyttäisi olevan myös XSLT-muunnosten kanssa [39]. Tehokkuustestien tulokset riippuvat tietenkin aina enemmän tai vähemmän käytettävästä aineistosta. Erityisellä laitteistona toteutetulla XSLT-muunnosprosessorilla voidaan suorituskyky viedä äärimmilleen.

Ensimmäinen XSLT-kääntäjä oli ja on Sun Microsystemsin sittemmin avoimen lähdekoodin Apache-projektille lahjoittama XSLTC. Se kääntää XSLT-dokumentit suoraan Java-tavukoodiksi. XSLTC-kääntäjää voidaan käyttää standardin JAXP-rajapinnan kautta. XSLT-dokumentit voidaan kääntää ennakkoon tai ajonaikaisesti, mikäli niihin halutaan tehdä muutoksia käynnistämättä sovellusta uudelleen. XSLTC ei tue vielä kaikkia XSLT 1.0:n ominaisuuksia, ja kehitys siltä osin voi olla hidasta, sillä painopiste on tällä hetkellä tehokkuuden parantamisessa [5].

XSLTC:n alkuperäinen suunnittelija, Jazek M. Ambroziak, lähti kehittämään uutta XSLT-kääntäjää Xalan/XSLTC 2.3.1 –version pohjalta [1]. Tämän kaupallisen Gregor-tuotteen kehittämisen tavoitteena on luoda tehokkain ja laadukkain XSLT-toteutus Java-ympäristöön. Gregor onkin menestynyt hyvin tehokkuusmittauksissa [39], missä sen suorituskyky on XSLTC-kääntäjään

verrattuna vähintään kaksinkertainen. Näihin mittauksiin on kuitenkin suhtauduttava erittäin varauksellisesti.

JSP-sivut käännetään aina ennen suorittamista, joten tehokkuusvertailu kääntämisen ja tulkkauksen välillä ei niiden osalta tule kyseeseen. JSTL-tagikirjaston toimintojen attribuuttien XPath ja Expression Language –lausekkeet kuitenkin käsitellään ajonaikaisesti. EL-lausekkeiden käyttö on luonnollisesti tehokkuutta heikentävä tekijä, verrattuna suoran Java-koodin käyttöön vastaavissa tilanteissa.

5.2.3 Käteismuistit

Käteismuisteja voidaan käyttää verkkopalvelujen yhteydessä useissa eri kohdissa koko palveluketjua. Tämän työn kannalta mielenkiintoisia ovat sovelluspalvelimen sisäiset XSLT-dokumenttien ja JSP-sivuista muodostettujen Servlettien sekä yleisten XML-dokumenttien käteismuistit.

JAXP-rajapinta tarjoaa luonnollisen keinon XSLT-dokumenttien käteismuistin toteuttamiseen. XSLT-tiedostoista muodostetut `Templates`-ilmentymät voidaan tallentaa muistinvaraisesti, jolloin XSLT-tiedostot ladataan ja jäsennetään vain kertaalleen. `Templates`-olioita voidaan käyttää rinnakkain useista eri säikeistä. Näistä luodaan erikseen `Transformer`-olio kutakin muunnosta varten [15]. `Templates`-olioiden käteismuistia voidaan hyödyntää riippumatta siitä, onko käytössä tulkkaava vai kääntävä XSLT-muunnosprosessori. XSLTC-kääntäjä tarjoaa mahdollisuuden toteuttaa myös lähdedokumenteille oman käteismuistin [6]. XSLTC käyttää kaikille XML-dokumenteille sisäistä, DOM-mallia muistuttavaa esitystapaa ja käteismuisti voidaan toteuttaa tehokkaasti pitämällä tähän muotoon jäsennettyjä dokumentteja muistissa.

JSP-sivuista muodostettujen Servlet-ilmentymien käteismuistin toteuttaa sovelluspalvelin. Oletusarvoisesti tietystä JSP-sivusta käytetään vain yhtä ilmentymää palvelemaan kaikkia sivulle kohdistuvia kutsuja, myös rinnakkaisesti [23]. JSP-tekniikkaan tai JSTL-kirjastoon ei liity mitään erityistä XML-lähdedokumenttien käteismuistia, joten sellainen joudutaan toteuttamaan itse tai ottamaan käyttöön esimerkiksi jokin tagikirjastototeutus.

5.2.4 Merkistömuunnosten ja jäsennyksen välttäminen

Javan sisäinen esitystapa merkkijonoille on 16-bittinen. Kun tässä muodossa oleva merkkijono kirjoitetaan selaimelle tiettyä merkistöä käyttävän `Writer`-ilmentymän kautta, täytyy jokainen merkki koodata merkistölle ominaisella tavalla. Tällaista koodausta ja XML-dokumentin ylimääräistä jäsentämistä esimerkiksi peräkkäisten muunnosten välissä on syytä välttää. Jäsennettäessä ulkopuolisesta järjestelmästä tulevaa XML-dokumenttia huolehtii XML-jäsennin binäärimuodossa olevan syötteen merkistön tulkinnasta.

5.2.5 Laiskuus XML-käsittelyssä

Laiskuus XML-käsittelyssä voi liittyä sekä XML-dokumentin jäsentämiseen että XSLT-muunnoksiin. Apache Xerces2 Java –jäsennin toteuttaa DOM-mallinsa *laiskasti*. DOM-puurakenteen solmuja ei aina käsitellä loppuun asti ennen kuin

niihin esiintyy viittauksia. Laiska käsittely on oletusarvoisesti käytössä mutta se voidaan haluttaessa ohittaa.

Schott ja Noga esittävät paperissaan *Lazy XML Transformations* [42] laiskan XSLT-tulkin, joka mahdollistaa satunnaiset haut XSLT-muunnoksen lopputulokseen. Tulospuun solmut tuotetaan vasta niitä tarvittaessa, joten sellaisissa tapauksissa, missä tarvitaan vain osa tulospuusta, saadaan toteutustavasta tehokkuusetua. Tällainen voi realisoitua ketjutetuissa muunnoksissa. Muunnosprosessorin käyttäjälle sen sisäinen laiska evaluointistrategia ei näy laskennan lopputuloksessa millään tavoin. Kehittäjien omien testien mukaan tämä XSLT-tulkki on tehokkaampi kuin muut yleisesti käytössä olevat ei-kaupalliset Java-pohjaiset XSLT-tulkit.

5.2.6 Best Practice –käytännöt

Yllä kuvattujen suorituskykyä parantavien käytäntöjen lisäksi XSLT:n käytöstä on dokumentoitu varsin runsaasti erilaisia ohjelmointikäytäntöjä, joiden tarkoituksena on parantaa sekä suorituskykyä että ylläpidettävyyttä. Esimerkiksi Dave Pawson on koonnut mittavan kokoelman XSLT:n käyttöön liittyviä kysymyksiä ja vastauksia [37] ja erityisesti XSLTC-kääntäjän käyttöön liittyviä tehokkuusnäkökohtia on listattu XSLTC:n ohjeistuksessa [6].

XSLT 1.0 sisältää yleisistä joukko-operaatioista ainoastaan unionin. Joukkojen leikkauksen ja erotuksen toteuttamiseksi voidaan käyttää Michael Kayn kehittämää puhtaasti XPath-lausekkeeseen perustuvaa metodia. Joitakin rekursiivisia kutsuja voidaan välttää muuttamalla ne iteraatioiksi Wendel Piezin kehittämällä algoritmilla. Kaksi vaihtoehtoa sisältäviä konditionaaleja voidaan nopeuttaa Oliver Beckerin metodilla ja ryhmittely voidaan toteuttaa tehokkaasti perustuen XSLT:n `key()`-funktioon Steve Muenchin kehittämällä metodilla [9].

JSP-tekniikan käyttöön liittyen pidetään yleisesti hyvänä käytäntönä hyödyntää tagikirjastotekniikoita. Erityisesti käyttämällä standardoitua JSTL-kirjastoa, voidaan jo sinällään hyödyntää tagikirjastojen kehittämisessä saatuja kokemuksia. Näin voidaan tuottaa selkeämpi ja siten helpommin ylläpidettävä lopputulos kuin käyttämällä Java-koodia suoraan JSP-sivuilla. Mihin JSTL ei tarjoa ratkaisuja, voidaan toteuttaa oma tagikirjasto tai hyödyntää olemassaolevia toteutuksia. JSP-sivut voidaan myös kirjoittaa kokonaan XML-dokumenteiksi [32]. Näin voidaan varmistua siitä, että tulostettava lopputulos on hyvinmuotoiltu XML-dokumentti ja muunnosten ketjuttaminen tai jopa lähdekoodin muokkaus onnistuu automatisoidusti.

5.2.7 Apache Tomcat –konfigurointi

Tomcat-palvelinohjelmiston konfiguroinnilla voidaan vaikuttaa sekä palvelimen yleisen toiminnan että JSP-sivujen käsittelyn tehokkuuteen. Keskeistä on asettaa virtuaalikoneen käytössä oleva olioiden säilyttämiseen käytettävän keko-rakenteen koko riittävän suureksi ja palvelimen käyttämien säikeiden määrä sopivaksi [28]. Säikeitä on oltava sen verran, että palvelimen maksimikuorma voidaan tyydyttää. HotSpot-virtuaalikoneesta kannattaa käyttää palvelinkäyttöön optimoitua versiota. Kaikenlaiset debuggaukseen liittyvät tulostukset on syytä

kytkeä pois tuotantokäytössä, ellei ole erikseen syytä olettaa, että niitä tarvittaisiin laajasti ongelmien selvittelyyn.

JSP-sivut käsittelee Tomcat-ohjelmistossa Jasper-kääntäjä. Sen toiminnallisuuteen liittyy mm. päivitettyjen sivujen uudelleenkääntäminen ja tagikirjastoluokkien ilmentymien käteismuisti (pooling). Jokaisella kutsulla tapahtuva tiedostojen aikaleimojen tarkistaminen on syytä kytkeä pois päältä ja poolaus vastaavasti päälle, mikäli haetaan erityistä tehokkuutta [3].

5.3 Vaihtoehtoja XML-käsittelyyn JSP-sivuilla

Tässä työssä keskitytään JSTL-kirjaston käyttöön XML-käsittelyssä JSP-sivuilla mutta paljon muitakin vaihtoehtoja on olemassa. JSTL:n yhteydessä ei tarvitse tuntea mitään muita, kuten JAXP, SAX tai DOM –rajapintoja. Näitä standardirajapintoja kuvattiin luvussa neljä.

dom4j [34] on yksi vaihtoehtoisista ratkaisuista XML-käsittelyyn. dom4j perustuu avoimeen lähdekoodiin. Lähtökohtana on ollut kehittää yksinkertainen, kaikki JAXP-rajapinnan tarjoamat mahdollisuudet sisältävä, erityisesti Java-ohjelmoijille intuitiivinen kirjasto XML-käsittelyyn. dom4j sisältää oman XML-jäsentimen ja XPath-prosessorin, mutta se voidaan konfiguroida käyttämään myös muita SAX-jäsentimiä JAXP-rajapinnan kautta.

XTags [2] on yksi varteenotettavista XML-tagikirjastoista ja JSTL:lle rinnakkainen Apache Taglibs –projektin osa. XTags-tagien toiminnallisuus tukeutuu dom4j-kirjastoon. Kirjaston XML-käsittelyn mahdollisuudet ovat huomattavasti JSTL:n xml-kirjastoa laajemmat ja muistuttavat läheisesti XSLT-kieltä. Jostain syystä nämä ominaisuudet eivät kuitenkaan ole löytäneet tietään JSTL-määritykseen.

Java-ympäristössä DOM ja dom4j-puumalleilla on kolmaskin vahva kilpailija; JDOM. Tämän rajapinnan ja kirjaston tavoitteet ovat samansuuntaiset dom4j:n kanssa. JDOM on Java Community Process –standardointiputkessa.

5.4 Arkkitehtuurimalleja

Zdun jakaa dynaamisen verkkopalvelun käyttöliittymän toteuttamistavat korkealla abstraktiotasolla kahteen eri malliin [55]. Esityspohjiin tai malleihin (template, page template) perustuvassa arkkitehtuurissa kirjoitetaan käyttöliittymäkoodi sellaisenaan ja sen joukkoon sisällytetään erityistä merkintäkieltä, jolla dynaamisesti tuotetut osat liitetään lopputulokseen. Useat web-käyttöliittymätekniikat, kuten XSLT, JSP, ASP ja PHP toteuttavat tämän arkkitehtuurin mukaisen ratkaisun. *Konstruktiivisessa* arkkitehtuurissa taas käyttöliittymän tuottaa *perinteinen* ohjelma. Tämän arkkitehtuurin toteuttaa esimerkiksi jo pitkään käytössä ollut CGI-rajapinta (Common Gateway Interface) ja J2EE-ympäristössä Servlet-luokat.

Esityspohjiin perustuvia ratkaisuja monimutkaisempia arkkitehtuurimalleja on esitetty useita. Alla oleva jako ei missään nimessä ole täydellinen luokittelu erilaisista malleista vaan pyrkii pikemmin tuomaan esiin muutamia

mielenkiintoisia ja ennen kaikkea toisistaan poikkeavia ratkaisutapoja. Näissä esiintyy piirteitä myös konstruktiivisesta lähestymistavasta. Monet mallit pyrkivät ratkaisemaan tai ainakin jättämään tilaa esimerkiksi monikanavaisuuteen ja personointiin liittyville ratkaisuille. Tällaiset ominaisuudet on rajattu tämän työn ulkopuolelle.

5.4.1 Suoraviivainen yhteiskäyttö

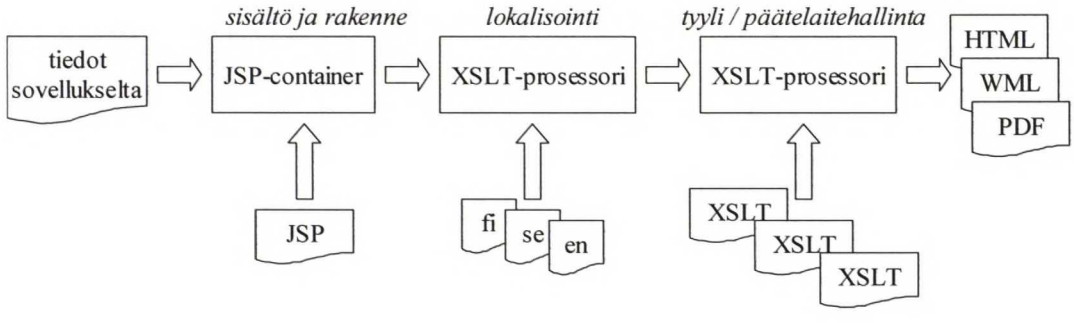
JSP-tekniikka tarjoaa käytännössä mahdollisuuden suorittaa mitä tahansa Java-koodia JSP-sivuilta käsin ja liittää siten syntyneen lopputuloksen osaksi JSP-sivun tulostusta. Tässä voidaan siten käyttää vaikkapa XSLT-tekniikkaa tuottamaan yksittäisiä osia sivun sisällöstä. JSTL-kirjaston toiminnot `x:parse` ja `x:transform` auttavat XML-dokumenttien jäsentämisessä ja XSLT-muunnoksen kutsumisessa. XSLT-tekniikan laajentaminen on mahdollista useilla eri tavoilla, kuten luvussa 3 todettiin. Tämä mahdollistaisi periaatteessa siis myös JSP-sivujen kutsumisen osana XSLT-muunnosta.

5.4.2 Ajonaikainen ketjutus

Bruchez ja Tazi kuvaavat artikkelissaan [14] Model-2X-arkkitehtuurin, jossa hyödynnetään sekä JSP-tagikirjastotekniikoita että XSLT-muunnoksia. JavaServer Faces (JSF) [50] on esitystapakehys, jonka keskeinen osa on joukko päätelaiteriippumattomia käyttöliittymäelementtejä. Nämä elementit päätelaitespesifillä tavalla tuottava kirjasto voidaan toteuttaa ns. RenderKit-kirjastona. RenderKit-kirjastoja kutsutaan JSF-kehykseen kuuluvan tagikirjaston avulla. Model-2X-arkkitehtuurimallin ideana on toteuttaa RenderKit, jolla voidaan tuottaa tagikirjastoa käyttäen yksinkertainen XML- tai XHTML-dokumentti. Tämä dokumentti otetaan syötteeksi sivun lopullisen ulkoasun tuottavalle XSLT-muunnokselle. Näin JSP-sivut ovat hyvin yksinkertaisia ja niillä riittää esittää vain sivun rakenne yleisesti ja päätelaiteriippumattomasti. Tyyliin vaikuttavat määrittelyt voidaan keskittää XSLT-muunnosdokumentteihin [14].

Yllä kuvatussa ratkaisussa ketjutettiin vain kaksi vaihetta, mutta useampiakin voidaan ketjuun lisätä. Bruchez ja Tazi esittävät esimerkkeinä tällaisista ketjun vaiheista lokalisoinnin eri kielille, erilaisten selainten ja päätelaitteiden tukemisen ja käyttöliittymän standardiosien lisäämisen [14].

Alla oleva kuva havainnollistaa mahdollisuuksia muunnosten ajonaikaiseen ketjuttamiseen:



Kuva 5-1: Hypoteettinen muunnosten ketjutus

XSLT-muunnosten välissä ei dokumenttia välttämättä tarvitse jäsentää uudelleen, sillä esimerkiksi Xalan-muunnosprosessoria käytettäessä dokumentti voidaan välittää seuraavaan vaiheeseen DOM-mallin mukaisena puurakenteena.

5.4.3 Käännösaikainen esikäsittely

Dunning kuvaa ratkaisun [20], jossa tuotetaan dokumentteja seitsemälle eri kielelle monivaiheisen, käännös- ja ajonaikaisen käsittelyn yhdistelmän avulla. Tässä ratkaisussa muunnostiedostot on aluksi jaettuina kuuteen eri kategoriaan. Esimerkiksi eri kieliset tekstit, yleisesti käytetyt mallit (template) ja kunkin dokumenttityypin runko ovat omina tiedostoinaan. Näistä tiedostoista kootaan XSLT-muunnoksella dokumenttityyppikohtaisesti haluttu dokumentti yhdistämällä kaikki tarvittavat osat ensin yhteen tiedostoon. Tämän jälkeen valitaan kieli ja toisella XSLT-muunnoksella korvataan tekstille varatut kohdat lokalisoidulla tekstillä. Näin siis tuotetaan kahdella peräkkäisellä, käännösaikaisella muunnoksella varsinainen valmiiksi lokalisoitu XSLT-tiedosto. Tätä dokumenttia käytetään ajonaikaisesti XML-dokumenttien muunnoksiin.

5.4.4 Sovelluskehyksistä

Sovelluskehys on joukko valmiita komponentteja, rajapintoja ja konfiguraatioita, jotka helpottavat sovelluskehitystä. Sovelluskehys määrää sovelluksen rakenteen, usein perustuen lukuisaan joukkoon erilaisia yleisesti hyviksi havaittuja suunnittelumalleja. Sovelluskehys toimii sovellukselle runkona ja kutsuu sovelluksen toiminnallisuutta.

Tässä työssä keskitytään käyttöliittymän tuottamiseen, joka on usein vain pieni osa sovelluskehyksen koko toiminnallisuutta. On olemassa myös erityisiä esitystavan (käyttöliittymän) tuottamiseen tehtyjä esitystapakehyksiä (presentation framework) [35]. Ne keskittyvät usein siihen, miten käyttöliittymä voidaan tuottaa erillään muusta sovelluksen toiminnallisuudesta ja silti kommunikoida sovelluksen kanssa mahdollisimman yksinkertaisella tavalla. Esitystapakehykset voivat sisältää esimerkiksi jonkin esityspohjiin (template) perustuvan mekanismin tai toteuttaa konfiguroitavan muunnosten ajonaikaisen ketjutuksen.

6. Luku - Ympäristö ja vaatimukset

Tässä työssä on hypoteettisena toimintaympäristönä verkkopalvelun käyttöliittymää tuottava J2EE-alustalla toimiva portaali. Tässä luvussa kuvataan tällaista verkkopalveluarkkitehtuuria, portaaliympäristöjä yleisesti ja työlle asetettuja reunaehtoja ja tavoitteita.

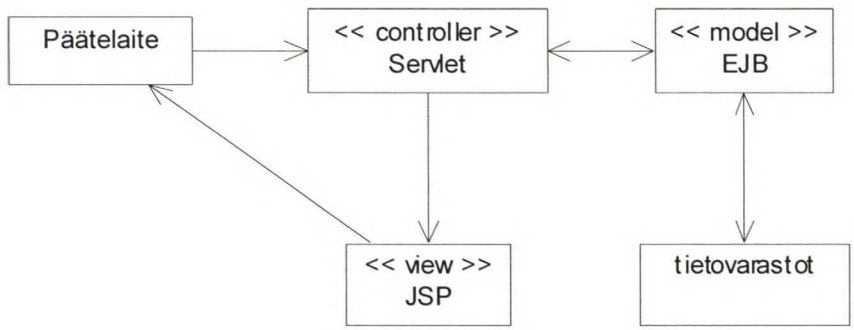
6.1 Dynaamiset verkkopalvelut

Yksinkertaisimmat verkkopalvelut ovat staattisia. Käyttäjälle tarjottava tietosisältö on kertaalleen julkaistu WWW-palvelimille, mistä sitä sellaisenaan siirretään selaimen näytettäväksi. Sisältö ja ulkoasu paketoidaan alusta alkaen samaan nippuun, Hyper-Text Markup Language (HTML) –kielellä kuvattuihin dokumentteihin. Staattisilla sivuilla toteutetu palvelut eivät kuitenkaan riitä pitkälle. Niinpä käyttöliittymää kuvaaville sivuille voidaan kirjoittaa palvelimen ymmärtämää sovelluslogiikkaa. Näin palveluihin saadaan dynaamisuutta esimerkiksi hakemalla käyttäjälle näytettävää tietosisältöä tietokannoista. Tässä käytettäviä tekniikoita ovat esimerkiksi CGI-rajapinnan (Common Gateway Interface) toteuttavat ohjelmat tai esityspohjiin perustuvat JSP ja ASP-tekniikat.

Laajempia palvelukokonaisuuksia kehitettäessä on havaittu hyödylliseksi ohjelmistokehityksen hallittavuuden, lopputuloksen ylläpidettävyyden ja toteuttajien selkeän vastuunjaon mahdollistamiseksi erottaa käyttöliittymä ja sovelluslogiikka omiksi komponenteikseen ohjelmistoarkkitehtuurissa. Käyttöliittymän kehittäjät voivat näin keskittyä näyttöjen tai sivujen ulkoasuun. Varsinaisten ohjelmoijien työksi jää suunnitella ja toteuttaa sovelluskomponentit, eikä heidän tarvitse tuntea käyttöliittymän toteuttamisen yksityiskohtia, saati graafista tai käytettävyyssuunnittelua. Tätä työnjakoa helpottaa työasemasovelluksissa jo pitkään näkynyt MVC-suunnittelumalli (Model-View-Controller). MVC erottaa toisistaan seuraavat vastuut eri komponenteille:

- Controller: käyttöliittymästä tulevien tapahtumien käsittely ja kutsujen ohjaaminen oikeille sovelluskomponenteille
- Model: sovelluslogiikkakomponentit ja niiden käyttämät sovelluksen tietovarastot
- View: käyttöliittymän muodostaminen sovelluslogiikan palauttamien tietojen perusteella

MVC:n muunnelma Model-2 on J2EE-ympäristössä yleisesti käytetty malli, joka soveltuu hyvin monimutkaisempien WWW-palveluiden tuottamiseen [43]. *Controller* voidaan toteuttaa Servlettinä, *model* JavaBean- tai EJB-komponentteina ja *view* JSP-sivuina:

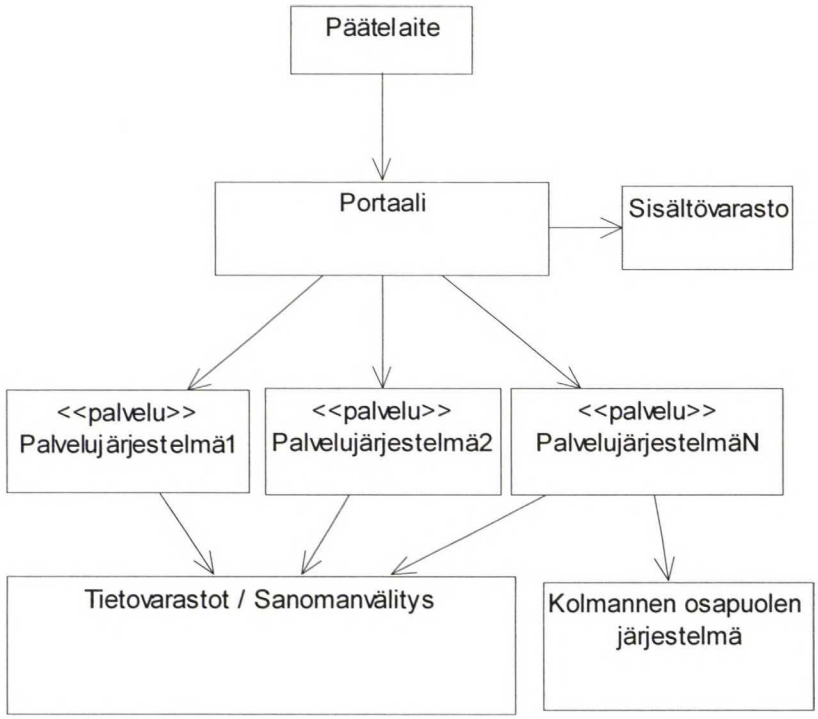


Kuva 6-1: Model-2-arkkitehtuuri

6.2 Monikerrosarkkitehtuuri

Sovelluspalvelimille toteutetut sovellukset on usein koottu varsin monimutkaisista kokonaisuuksista ja siten niissä usein käytetään monikerrosarkkitehtuuria (N-tier architecture). Monikerrosarkkitehtuurissa ohjelmisto jaetaan eri tehtäviä suorittaviin kerroksiin. Kerrokset voivat olla puhtaasti loogisia, ohjelmiston sisäisiä rakenteita. Yhtä hyvin toiminnallisuudet voidaan sijoittaa fyysisestikin erillisille palvelimille. Yksi mahdollinen malli on kolmikerrosarkkitehtuuri, jossa järjestelmä jaetaan käyttöliittymä-, palvelu- ja tietovarastokerroksiin. J2EE-ympäristössä käyttöliittymä- ja palvelukerroksen yhteistyö voidaan toteuttaa Model-2-mallin mukaisesti. Tietovarastokerroksessa voi olla hyvinkin monimutkaisia järjestelmiä, mistä voitaisiin tunnistaa lisää kerroksia.

Tämän työn hypoteettisena toimintaympäristönä on monikerrosarkkitehtuurin ylimmässä kerroksessa toimiva portaali. Alemmissa kerroksissa on palveluita, joiden kanssa viestitään XML-dokumentteihin perustuvalla protokollalla. Tämän työn kannalta ei ole merkitystä, millaisia järjestelmiä monikerrosarkkitehtuurin alemmissa kerroksissa tarkkaan ottaen on. Alla oleva kuva selventää tätä monikerrosarkkitehtuurin ajatusta:



Kuva 6-2: Työn toimintaympäristö: monikerrosarkkitehtuuri

6.2.1 Portaali ja portletit

Portaali laajassa merkityksessä on käyttäjälle ja päätelaitteelle räätälöity näkymä organisaation tarjoamiin palveluihin. Portaali voidaan ymmärtää hyvin laajasti, jolloin se tarjoaa paljon sellaisia mahdollisuuksia, mihin ei tässä työssä kuvattavissa ratkaisuissa oteta mitään kantaa. Portaalin yleisesti tarjoamia mahdollisuuksia voivat olla esimerkiksi seuraavat:

- personointi käyttäjästä kerätyn tiedon ja käyttäjän itse tekemien valintojen avulla
- mainoskampanjat
- käyttöoikeuksien hallinta
- kertakirjautuminen
- räätälöinti eri päätelaitteille
- räätälöinti eri selaimille

Osaa portaalin käyttöliittymässä esiintyvistä sisältöelementeistä kutsutaan *portleteiksi*. Ne muistuttavat jossain määrin ikkunointijärjestelmien ikkunoita. Kukin portlet-ohjelma on oma kokonaisuutensa ja muodostaa itse oman sisältönsä. Portletit voivat myös vaikuttaa toistensa toimintaan niin haluttaessa. Portletteja voi olla valmiiksi ohjelmoituna hyvinkin suuri joukko, joista portaali

näyttää käyttäjälle vain tietyt. Personointi, kampanjat ja käyttöoikeustiedot voivat vaikuttaa näytettävien portlettien valintaan ja sijoitteluun.

6.2.2 Palvelut ja palvelukeskeinen arkkitehtuuri

Portaalin ja palveluiden välillä viestitään XML-dokumenteilla. Palvelut toteuttavia järjestelmiä voi olla useita. XML-kutsuille ja vastauksille määritellään tietty rakenne ja protokolla. Sovelluslogiikka toteutetaan yleiskäyttöisinä palveluina, joita kutsumalla voidaan koostaa erilaisia sovelluksia. Esimerkiksi erilaisille päätelaitteille tarkoitetut käyttöliittymät voivat hyödyntää samoja palveluita, vaikka käyttöliittymäkerrokset olisi toteutettu aivan eri tekniikoilla.

Palveluiden kutsumisesta vastaa portaalissa tietty portlet-ohjelma. Sen voidaan ajatella toteuttavan yllä kuvatun MVC-mallin mukaisen arkkitehtuurin. Portlet olisi tässä *controller*-roolissa. Palvelujärjestelmä toteuttaisi MVC-mallin *model*-toiminnallisuuden. Lopputuloksen portlet-ohjelma välittää näytettäväksi *view*-komponentille, jonka toteutusta tässä työssä tutkitaan.

Portaalin ja palveluita tarjoavien järjestelmien yhteistoiminnassa on piirteitä palvelukeskeisestä arkkitehtuurista. Tämä SOA-arkkitehtuurimalli (Service-Oriented Architecture) on ohjelmisto- tai järjestelmäarkkitehtuuri, jonka tavoitteena on saavuttaa mahdollisimman löyhä riippuvuus (loose coupling) palveluiden ja niiden kutsujien välille. Palvelukeskeinen arkkitehtuuri pyrkii toteuttamaan löyhän riippuvuuden seuraavia periaatteita noudattamalla [24]:

- Palvelu on selvästi määritelty tehtävä, joka suoritetaan perustuen palvelun rajapinnassa välitettäviin tietoihin.
- Palveluilla on mahdollisimman vähän erilaisia rajapintoja ja ne ovat yksinkertaisia. Nämä rajapinnat ovat laajasti saatavilla ja useat toteutukset voivat tarjota samaa palvelua.
- Palvelun kutsujan ei tarvitse tietää tai välittää siitä, missä kutsuttava palvelu todellisuudessa sijaitsee tai miten se on sisäisesti toteutettu.
- Palvelukutsut sisältävät hyvin vähän tai eivät lainkaan palvelun suorittamiseen liittyviä teknisiä parametreja.

SOA:een liittyy usein palveluhakemisto. Palveluhakemistosta voidaan hakea palveluita kuvailevia tietoja ja niiden perusteella tehdä varsinainen palvelukutsu. Näin kutsujan ei etukäteen tarvitse tietää, missä kutsuttava palvelu todellisuudessa sijaitsee. Palvelukeskeinen arkkitehtuuri ei ole aivan uusi asia, mutta se on viime aikoina ollut esillä Web Services –tyyppisten ratkaisujen saatua näkyvyyttä ohjelmistoteollisuudessa.

6.3 Aiheen rajaus

Tässä työssä keskitytään palvelukerroksen palauttamien vastausten käsittelyyn käyttöliittymän tuottavassa kerroksessa, eli portaalissa. Tutkitaan eri ratkaisutapoja näiden XML-vastausten käsittelyyn ja tietojen liittämiseen osaksi portaalisovelluksen tuottamaa XHTML-pohjaista käyttöliittymää. Käytännössä

tämä tarkoittaa yhden portletin sisällön tuottamista. Portaali voi tuottaa käyttöliittymäänsä huomattavan määrän muutakin sisältöä, eikä tässä tutkittava tekniikka siten kata koko käyttöliittymää. Pienille, kannettaville päätelaitteille suunnattuihin käyttöliittymiin ei myöskään oteta tässä työssä mitään kantaa.

6.3.1 Vaatimuksia

Toteutustekniikan on oltava koeteltu ja käyttökelpoiseksi todettu. Tietoa, kokemuksia ja osaamista tekniikasta on oltava saatavilla nyt ja tulevaisuudessa. Ratkaisun on rakennuttava mahdollisimman pitkälle avoimien standardien varaan. Joissakin kohdin tästä vaatimuksesta voidaan kuitenkin joutua joustamaan.

Yllä kuvatuista syistä johtuen ei valittu eksoottisia tai vasta standardointivaiheessa olevia tekniikoita, vaikkakin joihinkin sellaisiin työssä viitataan. Ensisijaiseksi tekniikaksi valittiin XML-dokumenttien muunnoksiin erikoistunut XSLT (Extensible Stylesheet Language – Transformations). Vertailukohdaksi valittiin JSP (Java Server Pages) ja sen laajennus, JSTL-kirjasto (JSP Standard Tag Library).

Keskeisin vaatimus toteutustekniikalle on riittävä ilmaisuvoima XML-dokumenttien käsittelyyn ja XHTML-käyttöliittymän tuottamiseen.

Sivujen suuren määrän vuoksi on toteutustekniikan mahdollistettava nopea ja rutiininomainen kehitystyö. Tätä voidaan tukea tunnistamalla ja toteuttamalla yleiskäyttöisiä osia. Myös virheiden paikallistamisen on oltava sujuvaa.

Ratkaisun on oltava yksinkertainen siinä mielessä, ettei se nojaudu liian moneen erilaiseen tekniikkaan. Tämä helpottaa kehittäjien pääsemistä vauhtiin toteutustyössä. Tästä syystä esimerkiksi JSTL-ratkaisussa pyritään mahdollisimman paljon tukeutumaan juuri JSTL-toteutuksiin, toteuttaen toiminnallisuuksia muilla tai omatekoisilla tagikirjastoilla vain aivan välttämättömissä tilanteissa.

Omia laajennuksia on voitava tehdä ennalta arvaamattomiinkin tilanteisiin. Laajennusten on oltava kutsuttavissa yksinkertaisesti varsinaisesta muunnoksen määrittävästä dokumentista tai ohjelmasta.

Palvelinten kuormitustaso voi olla korkea, joten tekniikan on oltava myös tehokas. Suorituskyky pyritään maksimoimaan sellaisin keinoin, jotka eivät merkittävästi heikennä ratkaisun muita tärkeiksi katsottuja ominaisuuksia. Tässä ovat keskeisiä tekijöitä esimerkiksi käännökset ja dokumenttien käteismuistit. Ajonaikaista tehokkuutta mitataan ja tekniikoita arvioidaan myös tältä pohjalta.

Ratkaisun on mahdollistettava lokalisointi usealle kielelle, tehokkaasti ja helposti. Dokumenttimuunnoksiin on voitava liittää useampia dokumentteja, esimerkiksi hakemalla virheilmoituksia keskitetystä XML-muotoisesta parametristosta. Tällaiset dokumentit on voitava pitää kertaalleen jäsennettyinä, nopeasti saatavilla käteismuistissa.

6.3.2 Työn tavoitteet kohdeyrityksen näkökulmasta

Työn tärkeimpinä tavoitteina on kohdeyrityksen kannalta

- tuottaa tietoa prototyyppiratkaisuihin valittujen tekniikoiden ja mahdollisten muiden vaihtoehtojen soveltuvuudesta yllä kuvattuun tehtävään
- lisätä yleistä XML-, XSLT-, JSP- ja tagikirjastotekniikoihin liittyvää osaamista
- tuottaa ohjeistusta tekniikoiden käyttöön ja ratkaisumallien soveltamiseen

7. Luku - Ratkaisut

Tässä luvussa kuvataan käytetyt työkalut ja olennaisimmat toteutetut ratkaisut. Ratkaisut tehtiin kahdella tekniikalla, jotka olivat XML-muunnoksiin erikoistunut XSLT ja käyttöliittymien tuottamiseen tarkoitettu JSP, painopisteenä siihen liittyvä JSTL-kirjasto.

7.1 Työkalut

Java-ohjelmointi tehtiin Borland JBuilder 9 –kehitysvälineellä. Tähän sisältyi yleisten luokkakirjastojen, JSP-tagikirjastojen ja JSP-sivujen toteutuksia. XSLT ja muut XML-dokumentit kirjoitettiin Altova XMLSpy ja Cooktop –työkaluilla.

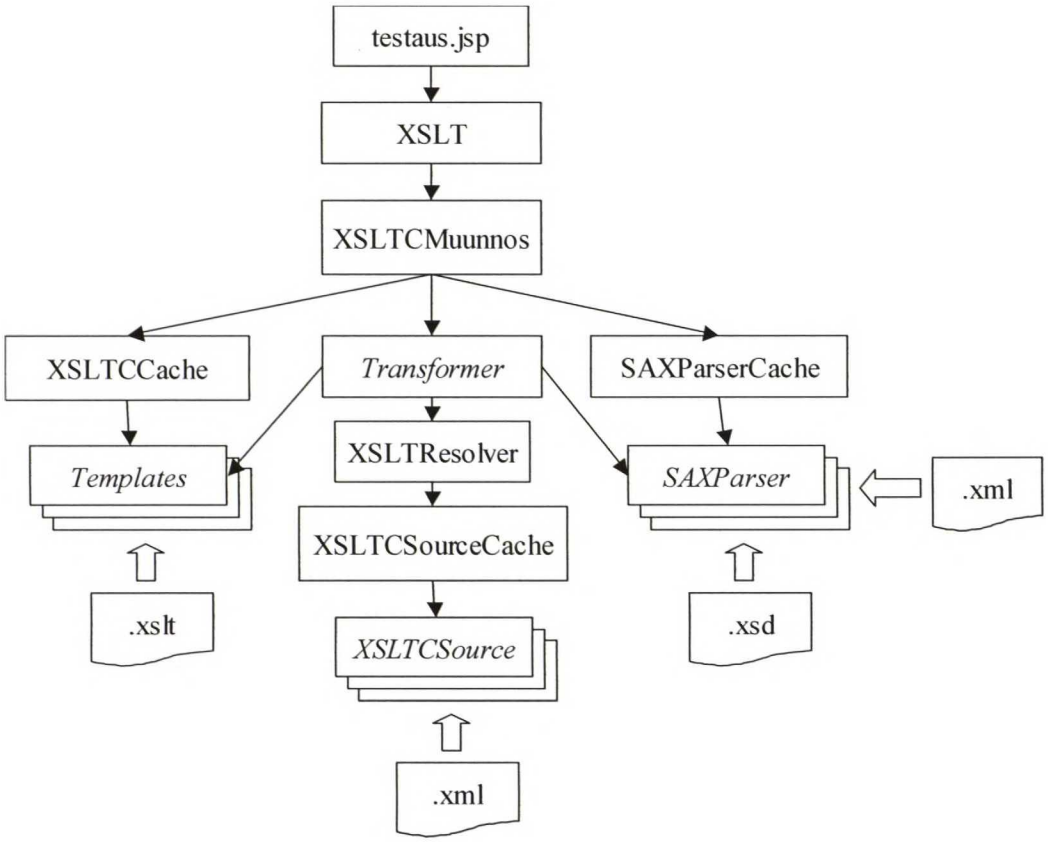
Testien ajamiseen käytettiin sovelluspalvelimena Apache Tomcat –ohjelmiston versiota 4.1.24. Java-virtuaalikone oli Sun Microsystemsin J2SDK 1.4.1_05-versioon sisältyvä HotSpot.

XML-jäsentimenä käytettiin Apache Xerces2 Java –jäsentimen versiota 2.4.0. XSLT-muunnosprosessorina oli Apache Xalan-Java 2.5.2 –version mukana toimitettava XSLTC-kääntäjä ja näiden yhteiset ajonaikaiset luokat. JSTL-tagikirjaston toteutuksena käytettiin Apache Taglibs –projektin toteutusta Standard 1.0.5. Tähän sisältyi ulkoisena Jaxen 1.0 –XPath-prosessori.

XSLT-tiedostojen käsittelyssä käytettiin lisäksi käännösprosessin kokeilemiseen Apache Ant –käännöstyökalua.

7.2 XSLT-ratkaisun peruskomponentit ja toiminta

Alla oleva kaavio kuvaa keskeisimmät *XML-käsittelyyn liittyvät luokat* ja niiden väliset riippuvuudet. Tässä mallissa mukana olevaa `SAXParser`-ilmentymien käteismuistia käytetään ainoastaan silloin, kun XML Schema-dokumentteihin perustuva validointi on kytketty päälle. Kaaviossa *kursiivit* kuvaavat standardirajapintojen toteutuksia, muut luokat ovat tässä työssä itse tehtyjä.



Kuva 7-1: XSLT-toteutuksen keskeiset komponentit ja riippuvuudet

7.2.1 XSLT-muunnoskomponentti

XSLT-muunnosten testaamista varten toteutettiin erillinen `XSLT`-muunnosluokka. Se piilottaa yksinkertaisen rajapinnan taakse kaiken muunnostoiminnallisuuden siten, ettei luokan kutsujan tarvitse tietää mitään käytettävästä XSLT-toteutuksesta. Muunnosluokan rajapinta ja konfigurointitiedosto kuvataan liitteessä *XSLT-muunnoskomponentti*.

7.2.2 XSLT-dokumenttien kääntäminen ja käteismuisti

XSLT-dokumentit käännetään Java-luokiksi Xalan-toteutukseen kuuluvalla XSLTC-kääntäjällä. Käännettyjä luokkia edustavat `Templates`-oliot pidetään käteismuistissa, jonka toiminnallisuuden toteuttaa `XSLTCCache`-luokka.

`XSLTCCache` voidaan parametroida joko kääntämään XSLT-tiedostoja ajonaikaisesti tai käyttämään vain valmiiksi käännettyjä luokkia. Asetus `cache_vain_classpath` määrää, haetaanko XSLT-luokkia ainoastaan normaalista Java-luokkapolusta vai haetaanko niitä myös erillisistä hakemistorakenteista XML-muodossa ja luokkatiedostoina. Jälkimmäisessä tapauksessa käteismuisti pollaa tiettyä xsl- ja class-tiedostoa levyltä aina kun asetuksella `cache_levypollaus` määritelty aika sekunteina on kulunut ko. muunnostiedoston osalta. Jos xsl-tiedosto on uudempi kuin edellisellä käännöksellä ja class-tiedostoa ei ole, käännetään xsl-tiedosto uudelleen.

Ajonaikaisen käännöksen etuna on, että XSLT-dokumentit voitaisiin vaihtaa milloin tahansa ja kääntää ne uudelleen osana verkkopalvelujärjestelmän normaalia toimintaa. Esikäännöksen etuna on ohjelmointivirheiden havaitseminen mahdollisimman varhaisessa vaiheessa. Esikäännöksiä voi tehdä joka tapauksessa, riippumatta siitä missä muodossa muunnosdokumentit siirretään tuotantoympäristöön.

7.2.3 XML-liitedokumenttien käteismuisti

Muunnoksessa tarvittavien XML-dokumenttien lataamisesta huolehtii JAXP-standardin mukaisen `URIResolver`-rajapinnan toteuttava luokka. Tämä `XSLTResolver` hoitaa kaikki XSLT:n `document()`-funktion kautta toteutettavat laajennukset. `XSLTResolver` ohjaa kaikki kutsut suoraan `XSLTCSourceCache`-luokalle, joka toteuttaa käteismuistin XML-dokumenteille. Käteismuistissa hyödynnetään XSLTC:n sisäisessä esitysmuodossa XML-dokumentteja käsittelevää `XSLTCSource`-luokkaa. Tällä ratkaisulla pystyttiin optimoimaan tehokkuutta välttämällä ylimääräistä XML-dokumenttien jäsentämistä.

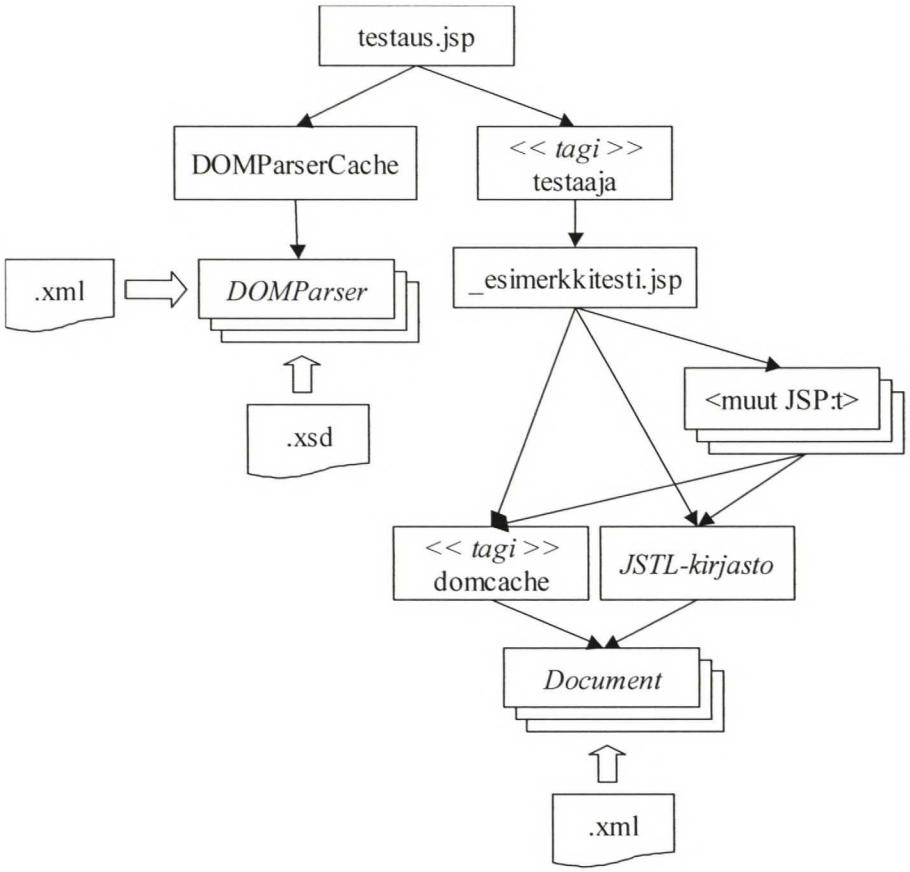
7.2.4 Lähdedokumenttien käsittely ja Schema-käteismuisti

XML-lähdedokumentit jäsennettiin suoraan XSLTC-kääntäjän sisäiseen esitysmuotoon. XSLTC käyttää tähän JAXP-rajapintojen kautta Xerces-jäsennintä.

XML Schema-dokumentteihin perustuva validointi voidaan myös kytkeä erikseen päälle, jolloin XSLT-muunnoskomponentti validoi lähdedokumentin jäsentäessään sitä Xercesin SAX-parserin avulla ennen varsinaista XSLT-muunnosta. Tarvittavat `SAXParser`-oliot pidetään käteismuistissa. Kukin parseri-ilmentymä tallentaa käyttämänsä XML Scheman muistinvaraisesti.

7.3 JSP-ratkaisun peruskomponentit ja toiminta

Alla oleva luokkamalli kuvaa keskeisimmät *XML-käsittelyyn liittyvät komponentit* ja niiden väliset riippuvuudet JSP-ratkaisussa. Kaaviossa *kursiivit* ovat standardirajapintojen toteutuksia. Muut luokat ovat tässä työssä itse toteutettuja. *tagi*-stereotyypillä kuvatut luokat ovat itse tehtyjä JSP-tagilaajennuksia.



Kuva 7-2: JSP-toteutuksen keskeiset komponentit ja riippuvuudet

7.3.1 JSP-sivujen käsittely

JSP-sivut käännetään Java-luokiksi Apache Tomcat –sovelluspalvelimeen kuuluvalla Jasper-kääntäjällä. Ohjelmiston perustoiminnallisuus mahdollistaa JSP-sivujen päivittämisen käynnistämättä palvelinta uudelleen. JSP-sivujen lähdekooditiedostojen ja käännettyjen luokkien aikaleimoja vertaillaan aika ajoin ja uudet versiot käännetään tarvittaessa.

JSTL-tagikirjastona käytetään Apache Taglibs-projektissa toteutettua Standard 1.0.5 –versiota. Tämä on riippuvainen ulkoisesta XPath-lausekkeiden ajonaikaiseen käsittelyyn erikoistuneesta kirjastosta ja sellaisena käytetään Jaxen 1.0 –toteutusta. Tagien käsittelyyn liittyy Tomcat-palvelimessa tagit toteuttavien ilmentymien uudelleenkäyttö (pooling).

JSP-toteutuksen testaamista varten käytetään itse toteutettua `tagit:testaaja`-tagia, jolla varsinaisen testattavan JSP-sivun tulostus voidaan ottaa talteen tai hävittää niin haluttaessa.

7.3.2 XML-dokumenttien käsittely ja käteismuistit

XML-lähdedokumentit jäsennetään Xerces2-Java-jäsentimellä ja niistä luodaan muistiin DOM-mallin mukainen `Document`-olio. XML Schema-dokumentteihin perustuvaa validointia voidaan käyttää haluttaessa. `DOMParser`-ilmentymät

pidetään käteismuistissa ja kullakin ilmentymällä on tallessa validoinnissa käytettävä XML Schema.

Liitedokumentit jäsennetään samalla tavoin kuin varsinainen XML-lähdedokumenttikin. Jäsentäminen ja haku kuitenkin tapahtuu itse toteutetun `DOMCache`-tagin avulla. Kyseinen tagi toteuttaa XML-dokumenteille käteismuistin. DOM-mallin mukaiset `Document`-oliot eivät ole käytettävissä eri säikeistä yhtä aikaa, joten tuotantokäyttöä varten käteismuistista olisi toteutettava synkronoitu versio.

7.4 Yleinen parametrointi

Molemmissa ratkaisuissa on huomioitu joukko ajonaikaisesti asetettavia parametreja. Näitä ovat seuraavat:

- `kuvapolku`: yleinen polku kaikille käyttöliittymään sisällytettäville staattisille kuville
- `ohjepolku`: yleinen polku erilaisille käyttöä ohjeistaville dokumenteille
- `kohde`: linkkien (HTML-ankkuri) ja lomakkeiden (HTML-form) kohteena käytettävän Servletin nimi
- `id`: käytettävää palvelua yksikäsitteisesti kuvaava tunniste, joka liitetään mukaan linkkeihin ja lomakkeisiin
- `kieli`: ISO-639-standardin mukainen kielikoodi, esim. `fi`, `sv` tai `en`
- `xmlcache`: polku yleisiin muunnoksissa tarvittaviin XML-dokumentteihin, joita ladataan aina tarpeen mukaan käteismuisteihin

7.4.1 XSLT-ratkaisu

Yleiset parametrit toteutettiin kaikkiin XSLT-tiedostoihin `xsl:include`-käskyllä liitettävänä erillisenä tiedostona:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl=http://www.w3.org/1999/XSL/Transform
  xmlns:xsldoc=http://www.bacman.net/XSLdoc
  exclude-result-prefixes="xsldoc">
  <xsldoc:author>Pete Hakkarainen</xsldoc:author>

  <xsl:param name="kuvapolku" select="'/images'" />
  <xsl:param name="ohjepolku" select="'/ohjeet'" />
  <xsl:param name="kieli" select="'fi'" />
  <xsl:param name="kohde" select="'foo.jsp'" />
  <xsl:param name="id" select="'bar'" />
  <xsl:param name="xmlcache" select="'file:///C:/xmlcache/'"/>

</xsl:stylesheet>
```

Parametreilla on vakioarvot, jotka voidaan kumota syöttämällä ne muunnosluokalle hajautusrakenteessa.

7.4.2 JSP-ratkaisu

Yleiset parametrit lisätään JSP-toteutuksessa `c:set`-toiminnolla erillisessä JSP-tiedostossa, joka sisällytetään mukaan päätiedostoon `include`-direktiivillä. Muuttujien sisältö kovakoodattiin tähän prototyyppiin:

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>

<c:set var="kuvapolku" value="/images" scope="request" />
<c:set var="ohjepolku" value="/ohjeet" scope="request" />
<c:set var="kieli" value="fi" scope="request" />
<c:set var="kohde" value="foo.jsp" scope="request" />
<c:set var="id" value="bar" scope="request" />
<c:set var="xmlcache" value="file:///c:/xmlcache/" scope="request" />
```

7.5 XHTML:n tuottamisen perusratkaisut

Staattisen XHTML-sisällön tuottaminen molemmilla toteutustavoilla on triviaalia. Tässä kuvataan miten ohjelmallista käsittelyä vaativa dynaaminen sisältö tuotetaan. Tärkeimpiä tähän kuuluvia rakenteita ovat linkit, lomakkeet, painikkeet, selaimella ajettavat skriptit sekä kuva- ja tyyliviittet.

7.5.1 XSLT-ratkaisu

Linkkien, lomakkeiden, painikkeiden, selaimella ajettavien skriptien ja tyyli- sekä kuvaviitteisen tuottaminen on suoraviivaista yllä kuvattujen parametrien avulla. Alla olevassa esimerkissä havainnollistetaan näitä:

```
<form action="{$kohde}?id={$id}" name="palvelu" method="post">
  <input type="submit" name="suorita_toiminto" value="Suorita" />
</form>

<a href="{$kohde}?id=uusi_palvelu"> Toiseen palveluun... </a>

<img src="{$kuvapolku}/yleinen_kuva.gif" alt="-" />

<img src="{$kuvapolku}/{$kieli}/lokalisoitava_kuva.gif">
    <xsl:attribute name="alt">
       <xsl:call-template name="lokalisoitava_kuva"/>
    </xsl:attribute>
</img>

<a href="javascript:avaaOhje({$ohjepolku}/ohje_x_.html)">
  <img src="{$kuvapolku}/ohje_kuva.gif" alt="Ohjeeseen" />
</a>

<script type="text/javascript">
  <xsl:comment>
function avaaOhje(dokumentti) {
  // avataan dokumentti samaan tai uuteen selainikkunaan
}
// piiloon vanhoilta selaimilta </xsl:comment>
</script>
```

7.5.2 JSP-ratkaisu

Myös JSP-ratkaisu on suoraviivainen. Huomattavaa on, että muuttujiin viittaamisessa joudutaan kirjoittamaan "ylimääräistä" koodia, sillä käytetty JSP-versio 1.2 ei vielä tue Expression Language-lausekkeita muualla kuin JSTL-tagien attribuuteissa. JSP-versio 2.0 sisältäisi mahdollisuuden käyttää EL-lausekkeita missä tahansa.

```
<form action="<c:out value="${kohde}"/>?id=<c:out value="${id}"/>"
      name="palvelu" method="post">
  <input type="submit" name="suorita_toiminto" value="Suorita" />
</form>

<a href="<c:out value="${kohde}"/>?id=uusi_palvelu">Toiseen palveluun...
</a>

<img src="<c:out value="${kuvapolku}"/>/yleinen_kuva.gif" alt="-" />

<img src="<c:out value="${kuvapolku}"/>/<c:out
          value="${kieli}"/>/lokalisoitava_kuva.gif"
     alt="<fmt:message key="kuvateksti"/>" >
</img>

<a href="javascript:avaaOhje(<c:out
    value="${ohjepolku}"/>/ohje_x_.html)">
  <img src="<c:out value="${kuvapolku}"/>/ohje_kuva.gif"
       alt="Ohjeeseen" />
</a>

<script type="text/javascript">
<!-
function avaaOhje(dokumentti) {
  // avataan dokumentti samaan tai uuteen selainikkunaan
}
// piiloon vanhoilta selaimilta -->
</script>
```

7.6 XML-käsittelyn perusratkaisut

Varsinaisen lähdedokumentin käsittely voidaan hoitaa hyvin pienellä joukolla erilaisia tekniikoita. Sekä XSLT- että JSTL-ratkaisuissa toistuvat samat yksinkertaiset rakenteet, vain hieman erilaisilla syntakseilla. Molemmat ratkaisut tukeutuvat vahvasti XPath-lausekkeisiin.

Käsittely molemmilla tekniikoilla noudattaa tässä *pull*-mallia. Lähdedokumentti ei siis ohjaa tulostusta vaan kontrolli on XSLT-dokumentilla tai JSP-sivulla. Todelliset käsiteltävät lähdedokumentit ovat hyvin erilaisia ja sisältävät liiketoiminnallista tietoa, eivät juurikaan käyttöliittymää kuvailevia rakenteita. Tämä tekisi *push*-tyyppisen käsittelyn toteuttamisesta vaikeaa ja toisaalta JSTL-ratkaisussa tällainen käsittely ei olisi niin suoraviivaisesti toteutettavissa kuin XSLT:llä.

7.6.1 XSLT-ratkaisu

Yksinkertaiset iteraatiot ja tietojen hakeminen toteutettiin XSLT-ratkaisussa perusmekanismeilla:

```
<xsl:template match="iteroitava_dokumentin_osa">
```

```
  <xsl:for-each select="iteroitava_taulukko/elementti">
    <option value="{@attribuutti}">
      <xsl:value-of select="."/>
    </option>
  </xsl:for-each>
</xsl:template>
```

7.6.2 JSP-ratkaisu

Yksinkertaiset iteraatiot ja tietojen hakeminen toteutettiin JSP-ratkaisussa JSTL:n xml-osakirjaston tageilla:

```
<x:forEach select="$requestScope:xmlsanoma/iteroitava_taulukko">
  <option value="<x:out select="@attribuutti" />">
    <x:out select="."/>
  </option>
</x:forEach>
```

7.7 Lokalisointi

Lokalisointi eli käyttäjän maantieteellisen sijainnin ja kielen mukaan tapahtuva dynaaminen sisällön tuottaminen on ratkaisuissa erityisen tärkeä. Se tehdään molemmissa toteutuksissa ajonaikaisesti ja toteutustavan on oltava tehokas. Tässä ei huomioida käyttäjän maata vaan ainoastaan kieli ISO-639-standardin mukaisina kielikoodeina. Ratkaisut on tehty prototyypeissä vain kaksikielisinä, mutta ne ovat laajennettavissa suoraviivaisesti useammalle kielelle.

7.7.1 XSLT-ratkaisu

XSLT ei suoraan tue lokalisointia. Eri kieliset tekstit ovat toteutetussa ratkaisussa kutsuttavina malleina erillisessä XSLT-tiedostossa, joka otetaan mukaan `xsl:include`-käskyllä. Tämä on testatuista ratkaisuista tehokkain, mutta tekstit sisältävässä tiedostossa on jonkin verran "turhaa" koodia. Ratkaisun muita vahvoja puolia ovat ne, että se toimii suoraan millä tahansa XSLT-prosessorilla eikä vaadi mitään esikäsittelyä. Ajonaikaisuus tuo toki hieman lisäkuormaa suoritukseen.

Esimerkki XSLT-tiedostosta, jossa sisältöä lokalisoidaan:

```
<xsl:include href="tekstit/tekstit_esimerkki.xsl" />
<xsl:template match="/" />
  <h1><xsl:call-template name="t_otsikko1" /></h1>
</xsl:template>
```

Vastaavan lokalisointitiedoston `tekstit/tekstit_esimerkki.xsl` `xsl:template`-määrittely:

```
<xsl:template name="t_otsikko1">
  <xsl:choose>
    <xsl:when test="$kieli = 'fi'">Otsikko suomeksi</xsl:when>
    <xsl:when test="$kieli = 'sv'">Otsikko ruotsiksi</xsl:when>
  </xsl:choose>
</xsl:template>
```

Muut testatut ratkaisut perustuivat oikean tekstin hakemiseen XPath-hahmojen avulla. Lokalisoituja tekstejä haettiin XSLT- ja XML-dokumenteista. Tällä tavoin kutsut olivat kuitenkin monimutkaisempia ja ratkaisut hitaampia kuin yllä kuvattu.

7.7.2 JSP-ratkaisu

JSTL tarjoaa lokalisointiin "standardiratkaisun", sillä yksi JSTL:n neljästä tagikirjastosta on erikoistunut lokalisointiin ja tietojen muotoiluun. Tämä mekanismi on suoraviivainen ja kehittäjille mahdollisesti ennestään tuttu. Tässä ratkaisussa ei käytetä XML:ää lainkaan, kuten XSLT-lokalisoinnissa tehtiin.

JSP-sivulla asetetaan kielikoodista tai kieli- ja maakoodeista muodostuva `locale`, tekstit sisältävä `properties`-tiedosto (`fmt:setBundle`) ja haetaan tiedostosta arvoa avaimella:

```
<fmt:setLocale value="fi" />
<fmt:setBundle basename="kokeilu" scope="request" />
<fmt:message key="lokalisoitu_arvo" />
```

properties-tiedosto `kokeilu_fi.properties` asennetaan johonkin Java-luokkapolussa olevaan hakemistoon (esim. web-sovelluksen sisällä `/WEB-INF/classes`). properties-tiedoston sisältö koostuu nimi-arvo-pareista:

```
nimi=arvo
t_otsikko1=Otsikko suomeksi
```

J2EE-sovelluspalvelimilla on mahdollisuus puskuroida tällä tavoin haettavia tietoja muistiin mutta tämä ei ole standardin mukaan välttämätöntä. Testisivulla tiedot haettiin vain kertaalleen, jottei tämä vaikuttaisi testituloksiin.

7.8 Yleiskäyttöiset osat

Usein esiintyvien toiminnallisuuksien tunnistaminen ja toteuttaminen aidosti yleiskäyttöisiksi on haastavaa, mutta pidemmän päälle tehokkuusetuja tuovaa toimintaa. Tässä pyrittiin tunnistamaan yleisiä toiminnallisuuksia ja standardoimaan toteutustavat niiden osalta. Ensimmäinen tunnistettu yleinen toiminnallisuus oli muunnosten parametrointi, jota jo yllä kuvattiin. Muita yleistettäviä toiminnallisuuksia ovat esimerkiksi erilaisten tietotyyppien muotoiluun käytettävät kirjastot ja tietyn tyyppisten toistuvien käyttöliittymärakenteiden tulostamiseen tehdyt parametroitavat yleisosat.

Mahdollisimman laajan yleistämisen aikaansaamiseksi pitäisi käsiteltävien XML-dokumenttien noudattaa yhtenäistä rakennetta ja olla nimeämiseltään ja metatiedoiltaan samanlaisia. Tämä ei prototyypeissä toteutunut ja niinpä yleistämismahdollisuudet olivat vain kohtalaiset.

7.8.1 XSLT-ratkaisu

XSLT:llä yleisosat ovat toteutettavissa lähinnä yleiskäyttöisinä malleina (template). Yleisosien kutsumiseen on selvästi kaksi erilaista vaihtoehtoa:

- `xsl:call-template name="nimi"`, kun kontekstina on jo valmiiksi käsiteltävä solmu
- `xsl:apply-templates mode="moodi" select="elementti"`, jos halutaan valita käsiteltävä solmu tai solmujoukko erikseen

Molemmilla mekanismeilla voidaan liittää kutsuihin myös parametreja `xsl:with-param`-elementeillä. Yleisosat otettiin mukaan `xsl:include`-käskyllä ja kutsumekanismiksi valittiin `xsl:apply-templates`. Esimerkkikoodia päivämäärän muotoilevasta yleisestä mallista on liitteessä *Päivämäärän muotoilu*.

Jos olisi ollut mahdollista hyödyntää käsiteltävien XML-dokumenttien elementtien nimeämistä ja metatietoja tehokkaasti, olisi mahdollisesti parempi tapa käyttää tässä *push*-mallin mukaista kutsumekanismia ja siis `xsl:apply-templates`-elementtiä sopivalla kontekstisolmulla. Tällöin sovellettavan mallin valinta jäisi XSLT-prosessorille.

7.8.2 JSP-ratkaisu

Yleiskäyttöisiä JSP-sivujen osia voidaan erottaa omiksi JSP-tiedostoikseen. Niiden sisällyttämiseen runkotiedostoihin on käytännöllistä kahdella eri tavalla. `@include`-direktiivillä yleisosia voidaan sisällyttää siten, että tiedostojen yhdistäminen tapahtuu ennen käännöstä. `jsp:include`-toiminnolla voidaan kutsua toista JSP-sivua ajonaikaisesti.

JSP-sivujen yleiset parametrit ja virheilmoitusten käsittely ovat osa jokaista runkosivua, joten ne sisällytetään käännösaikaisesti `@include`-direktiivillä. `jsp:include`-toimintoa käytetään ainoastaan testattavan sivun kutsumiseen suorituskykyä mittaavalta sivulta. JSTL-kirjastoon kuuluvaa `c:import`-tagia ei ratkaisussa käytetä.

Yleisosia on JSP-sivuilla kätevä toteuttaa omina tagikirjastoina JSP-palasten sijaan ja tätä tapaa käytetään ratkaisussa laajasti. Mekanismia kuvataan alla kohdassa *Omat laajennukset*.

7.9 Omat laajennukset

Omilla laajennuksilla tässä tarkoitetaan sellaisia toiminnallisuuksia, jotka eivät ole suoraan tuettuina XSLT tai JSTL-tekniikoissa ja joita varten toteutetaan omia Java-kirjastoja. Mahdollisia laajennusten kutsumekanismeja on molemmissa tekniikoissa useita ja tässä etsitään ratkaisuja, jotka toimisivat eri XSLT-prosessorien ja JSTL-tagikirjastojen toteutuksilla. Yksi esimerkki omista laajennuksista on virheilmoitusten tulostaminen ja XML-dokumenttien käteismuistin käyttö tässä.

7.9.1 XSLT-ratkaisu

Periaatteeksi otettiin, että XSLT-toteutus pyritään pitämään mahdollisimman riippumattomana käytettävästä XSLT-prosessorista. Tästä seuraa, että laajennusten toteutukseen käytetään yleisempiä JAXP-rajapintoja eikä Xalanin tai XSLTC:n omia mekanismeja.

Koska laajennusten toteuttaminen XSLT-elementteinä tai –funktioina on tehtävissä vain Xalanin natiivirajapintojen kautta, käytetään tässä laajennusten toteuttamiseen JAXP-standardin määrittelemää `URIResolver`-rajapintaa. Rajapintaan toteutettiin oma luokka `XSLTResolver`, jonka tehtävänä on välittää XSLT:n `document()`-funktiolla tehdyt XML-dokumenttien pyynnöt eteenpäin dokumenttikäteismuistille.

XML-dokumentteja voidaan ladata dokumenttikäteismuistista seuraavalla tavalla:

```
<xsl:variable name="dokumentti"
  select="document(concat($xmlcache, 'dokumentti.xml'))/juurielementti"
/>
```

Samalla mekanismilla olisi toteutettavissa mitä tahansa laajennuksia määrittämällä omia "protokollia" erilaisten laajennusten kutsumiseen:

```
<xsl:variable name="summa"
  select="document('omat://summa?x=1&y=1)"
/>
```

7.9.2 JSP-ratkaisu

JSP-sivuilta voidaan kutsua ulkoisia Java-luokkia pääasiassa kahdella tavalla; suoraan skriptleteistä tai tagikirjastojen kautta. Tässä ratkaisussa otettiin peruslinjaukseksi käyttää mahdollisimman paljon tagikirjastotekniikkaa JSP-sivujen ylläpidettävyyden parantamiseksi. Omia tageja toteutettiin viisi.

`tagit:domcache` –tagi toteuttaa XML-dokumenttien käteismuistista haun. Alla olevassa esimerkissä haetaan liitedokumentti XML-lähdedokumentissa olevan viitteen ja XML-dokumenttikäteismuistin sijaintitiedon perusteella. Haettu dokumentti tallennetaan DOM-mallin mukaisena `Document`-oliona sivun kontekstin muuttujaan `dokumentti`:

```
<x:set var="_uri" scope="page"
  select="string(concat($xmlcache,
                        $requestScope:xmlsanoma/taulukko_2/@viittaus,
                        '.xml'))" />
<tagit:domcache var="dokumentti"
                uri="<%= (String)pageContext.getAttribute("_uri") %>"
/>
```

`tagit:muotoileTilinumero` toteuttaa tilinumeroiden muotoilun. Se ottaa parametrinaan tilinumeron merkkijonona ja olettaa tiedon olevan oikeassa muodossa. Tässä ratkaisussa syntyy koodia JSP-sivulle tarpeettoman paljon:

```
<x:set var="formatoitava" scope="page"
       select="$requestScope:xmlsanoma/tilinro_1" />
<x:set var="formatoitava_tili" scope="page"
       select="string($pageScope:formatoitava)" />
<x:choose>
  <x:when select="$pageScope:formatoitava/@syottovirhe = '1'">
    <x:set var="tili" scope="page"
           select="string($pageScope:formatoitava/@arvo)" />
  </x:when>
  <x:otherwise>
    <tagit:muotoileTilinumero var="tili" tili="<%=
      (String)pageContext.getAttribute("formatoitava_tili") %>" />
  </x:otherwise>
```

```
</x:choose>
```

`tagit:muotoileDesimaali` toteuttaa desimaalilukujen muotoilun. Se ottaa parametrinaan suoraan DOM-mallin mukaisen rakenteen ja käsittelyä JSP-sivulla vaaditaan huomattavasti vähemmän kuin edellä kuvatussa tapauksessa. Tämä ratkaisu on kuitenkin riippuvainen käytettävän JSTL-kirjaston toteutuksen sisäisestä XML-käsittelymallista, tässä tapauksessa Apache-projektissa toteutetusta W3C:n DOM-malliin perustuvasta ratkaisusta:

```
<x:set var="formatoitava" scope="page" select="desimaalilukutieto" />
<tagit:muotoileDesimaali desimaaleja="4" desimaali="<%=
  (java.util.ArrayList)pageContext.getAttribute("formatoitava") %>"
/>
```

`tagit:yhdistaTaulukot` yhdistää kaksi DOM-rakenteena esitettävää taulukkoa ja palauttaa uuden DOM-rakenteen. Tämän toteuttaminen ei onnistunut JSTL-kirjaston XML-tagien avulla, mikä on katsottava puutteeksi. Tässäkin ratkaisussa sitoudutaan Apache-toteutuksen sisäiseen XML-esitystapaan.

`tagit:testaaja` mahdollistaa nopeustestien ajamisen. Sen avulla voidaan jättää tulostamatta tagin sisällä olevien JSP-elementtien tuottama tulostus.

7.10 Dokumentointi ja koodin kommentointi

XSLT-dokumenttien ja JSP-sivujen dokumentointi ja kommentointi lienee käytännössä hyvin vähäistä. Erityisesti yleisesti käytettävät osat kannattaisi kuitenkin dokumentoida käytön helpottamiseksi. Myös monimutkaiset ohjelman osat kannattaa kommentoida. Dokumentointi- ja kommentointikäytäntöjen on oltava selkeitä ja yksinkertaisia, muuten ne jäävät käytännössä tekemättä. Tästä syystä valittiin käytettäväksi tapa, jossa koodi kommentoidaan siten, että siitä voidaan haluttaessa luoda ohjelmallisesti dokumentaatiota.

7.10.1 XSLT-ratkaisu

XSLT-dokumenttien kommentointiin valittiin yksinkertaisuuden vuoksi XML-kommentteihin perustuva ratkaisu. XSLDoc-työkalulla voidaan tällaisista kommenteista luoda haluttaessa dokumentaatio, joka muistuttaa JavaDoc-työkalulla luotua Java-ohjelmien HTML-pohjaista dokumentaatiota. Vähintään seuraavat elementit olisi hyvä kommentoida: `xsl:stylesheet`, `xsl:template` sekä `xsl:param` ja `xsl:variable` juuritasolla. XSLDoc-työkalu tulkitsee tiettyjä `xsl`-nimiavaruuden elementtejä edeltävät XML-kommentit ja lisäksi kolme erilaista `xsldoc`-nimiavaruuden tagia, joista tässä esimerkissä esiintyy vain `xsldoc:author`. Esimerkki:

```
<!-Testistylesheetti XSLDocin demoamiseen -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xsldoc="http://www.bacman.net/XSLdoc"
                exclude-result-prefixes="xsldoc" version="1.0">
<xsldoc:author>Pete Hakkarainen</xsldoc:author>

<!-mikä tämä muuttuja -->
<xsl:variable .... />

<!-tekee sitä ja tätä ... <br>
```

```
    palauttaa ... <br>
    olettaa kontekstin olevan elementti ... -->
<xsl:template .... >
  <!-mikä tämä parametri on -->
  <xsl:param ... />
  ... templaten sisältö ...
</xsl:template>
```

7.10.2 JSP-ratkaisu

JSP-sivujen osalta kommentointi jätettiin vähäisemmäksi. Monet toiminnallisuudet ovat yleistettävissä omiksi tagikirjastoikseen ja tässä kannattaakin keskittyä niiden dokumentointiin. Tagikirjastot koostuvat normaaleista Java-luokista, joten dokumentoinnissa voidaan suoraan hyödyntää yleisesti käytettyä JavaDoc-kommentointia. JavaDoc on niin laajalti käytössä Java-kehittämisessä, ettei sitä tässä ole tarpeen kuvata.

7.11 Työkalut virheiden etsinnässä

XSLT-dokumenttien ja JSP-sivujen työstämiseen on olemassa lukuisa määrä erilaisia kehitysvälineitä. Näihin ei voitu, eikä ollut tarkoituskaan, tutustua tässä kovin laajasti. Yritettiin kuitenkin selvittää, miten näillä työkaluilla onnistuu virheiden etsintä ja korjaaminen, sillä se on kohdeyrityksessä havaittu ongelmalliseksi kohdaksi.

7.11.1 XSLT-ratkaisu

XSLT-dokumentteja tuotettiin XMLSpy ja XML Cooktop-kehitysvälineillä. Molemmat osaavat käyttää ulkoisia XSLT-muunnosprosessoreita ja XMLSpy:ssa on lisäksi oma sisäänrakennettu prosessorinsa.

Perinteinen *breakpointteihin* perustuva debuggaus onnistuu XMLSpy:n sisäänrakennetulla prosessorilla. Se on kuitenkin mahdotonta sellaisille XSLT-dokumenteille, jotka ovat riippuvaisia Xalan tai XSLTC-toiminnallisuudesta. Tällöin on kytkettävä päälle joko Xalanin sisäänrakennettuja debug-tulosteita tai käsiteltävä kehitettävä XSLT-dokumentti ensin sopivalla muunnoksella, minkä jälkeen se itse tuottaa debug-tulosteita muunnoksen yhteydessä. Testaamisessa käytettiin mm. Oliver Beckerin `trace.xsl`-dokumenttia debug-tulosteiden tuottamiseen [7].

Molemmilla yllä kuvatuilla tavoilla saadaan aikaan runsaasti tilannetietoa XSLT-muunnoksen etenemisestä ja virheiden pitäisi näin olla selvitettävissä. Mitään vaikeita ongelmatilanteita ei kuitenkaan prototyyppiratkaisuja kehitettäessä tullut vastaan, joten tällaisten tilanteiden selvittelyn mahdollisuuksiin ei voida tässä ottaa kantaa.

7.11.2 JSP-ratkaisu

JSP-sivut toteutettiin JBuilder 9 –kehitysvälineellä. JBuilder tarjoaa mahdollisuuden asettaa breakpointteja suoraan JSP-sivulla olevaan Java-koodiin. Toiminta perustuu Java Community Processissa määriteltyyn standardiin JSR-045

[51]. Myös omien tagikirjastojen virheiden etsintä onnistuu erinomaisesti, sillä tällaiset kirjastot voidaan ottaa projektiin mukaan lähdekoodina.

Ulkopuolisten, valmiiksi käännettyjen tagikirjastojen käyttöön liittyvä virheiden etsintä on ongelmallista, sillä esim. toimintojen (tagien) kutsuihin ei voida asettaa breakpointteja.

Lisäksi JBuilder piilottaa JSP-käännöksiin käytetyn Jasper-kääntäjän tuottamat tagikirjastojen kutsuihin liittyvät käännösaikaiset virheet. Tältä voidaan välttyä kääntämällä JSP-sivut vasta ajonaikaisesti, jolloin Jasperin tuottamat virheilmoitukset ovat selkeitä.

7.12 Yleisiä huomioita ja ongelmia

7.12.1 XSLT-ratkaisu

XSLT-ratkaisussa pyrittiin riippumattomuuteen XSLT-muunnosprosessorista. Tätä ei kuitenkaan täysin saavutettu. XSLT-mallit tuottavat lopputuloksenaan *result tree fragment* –tyyppistä tietoa, mikä ei ole enää käsiteltävissä esimerkiksi XPath-hahmojen avulla. Jatkokäsittelyä varten tarvitaan muunnos `node-set`-tyyppiseksi, mikä vaatii prosessorikohtaisen `node-set()`-laajennusfunktion käyttöä. Tämä funktio on määritelty osana EXSLT-laajennuskirjastoa, joten tästä ei kuitenkaan seuraa riippuvuutta suoraan käytettyyn muunnosprosessoriin vaan yleisesti sellaisiin muunnosprosessoreihin, jotka toteuttavat EXSLT-määrittelyn mukaisia laajennuksia.

Koska EXSLT-riippuvuus syntyi, käytettiin EXSLT-laajennuksia muuhunkin. Esimerkiksi päivämäärien muotoiluun ja merkkijonojen käsittelyyn toteutetut EXSLT-funktiot ovat potentiaalisia ehdokkaita käytettäviksi suorituskyvyn parantamisen nimissä.

Ratkaisussa pyrittiin yksinkertaisuuteen. Mitään esikäsittelyä tai peräkkäisiä muunnoksia hyödyntäviä ratkaisuja ei tehty. Tällä saadaan kehittäminen ja virheiden etsintä yksinkertaisemmaksi ja koko XSLT-muunnos, eikä vain osa siitä, on kehittäjän hallinnassa. Luonnollisesti tästä seuraa myös se, että kehittäjällä on oltava laaja osaaminen koko tekniikasta.

Xalan 2.5.2:n kanssa esiintyi ongelmia UTF-8-muotoisten tiedostojen käsittelyssä ja erikoismerkkien tulostamisessa. Samaan pakettiin kuuluvassa XSLTC-kääntäjässä vastaavia ongelmia ei ollut. Kokeiltaessa versiolla 2.6.0 lakkasi kuitenkin XSLTC:n sisäiseen esitystapaan (`XSLTCSource`-luokka) perustuva XML-käteismuisti toimimasta.

7.12.2 JSP-ratkaisu

JSP-ratkaisussa pyrittiin riippumattomuuteen JSTL-standardikirjaston toteutuksesta. Tämä ei kuitenkaan käytännössä toteutunut. Jouduttiin hyödyntämään Apache-projektin JSTL-toteutuksen sisäistä XML-esitystapaa, joka tosin on W3C:n DOM-mallin mukainen. Tämä tarve syntyi samassa kohtaa kuin XSLT-ratkaisussakin. Taulukoiden yhdistäminen monimutkaisella logiikalla ja

palauttaminen JSP-sivun käyttöön DOM-muodossa oli siis toteutettava JSTL-toteutukseen sidotulla tavalla.

Käytetty XPath-prosessori Jaxen ei tukenut XPath-standardin mukaista `lang()`-funktiota, minkä vuoksi XPath-lausekkeet jouduttiin kirjoittamaan elementin kielen tarkastelun osalta eri tavalla kuin XSLT-ratkaisussa. Jaxenilla oli ongelmia myös XML-nimiavaruuksien kanssa; `xsi:nil` –attribuutin tutkiminen lähdedokumenteista ei yksinkertaisesti onnistunut.

JSTL-kirjastoa pyrittiin käyttämään mahdollisimman paljon ja siten välttämään suoraa Java-koodin kirjoittamista JSP-sivuille tai muiden tagikirjastojen käyttöä. Tämä osoittautui hankalaksi lähestymistavaksi joidenkin XML-käsittelytilanteiden osalta. JSTL:n XML-käsittelymahdollisuudet ovat näiden kokemusten perusteella melko heikot ja niinpä parempi lähestymistapa voisi löytyä XML-käsittelyyn erikoistuneista tagikirjastoista tai toteuttamalla oma kirjasto juuri tämän käyttötapauksen vaatimusten mukaisesti.

8. Luku - Mittaukset ja analyysi

8.1 Ratkaisujen arviointi kehittämisen näkökulmasta

8.1.1 Osa-alueiden analyysi

Dynaaminen XHTML-käyttöliittymän tuottaminen onnistuu sekä JSTL- että XSLT-tekniikoilla mutkattomasti. Joissakin paikoin koodia syntyy tarpeettoman paljon, mutta se ei merkittävästi haittaa kehitystä. JSP 2.0 toisi tässä parannusta JSTL-ratkaisuun, sillä siinä yksinkertaiset Expression Language –lausekkeet ovat käytettävissä missä tahansa, eivätkä ainoastaan JSTL-tagien yhteydessä.

XML-käsittelyn perusratkaisut eli yksinkertaiset iteroinnit, ehdollinen käsittely ja tietojen hakeminen *pull*-mallin mukaisesti ovat molemmilla tekniikoilla hyvin samanlaiset. JSTL onkin perinyt nämä selvästi XSLT:ltä ja pyörää ei ole tässä keksitty uudelleen. Ratkaisut ovat toimivia.

Yhtenä periaatteena ratkaisuissa oli mahdollistaa suoraviivainen kehitystyö ilman erillisiä esikäännöksiä tai –muunnoksia. Tämä onnistuu hyvin molemmilla tekniikoilla eikä aiheuta liikaa lisäkuormaa suoritusaikaisesti. JSP/JSTL on tässä suhteessa parempi sillä toteutus voidaan tehdä millä tahansa JSP-sivujen kehitystä tukevalla Java-kehitysvälineellä. XSLT-dokumenteissa joudutaan toisinaan turvautumaan XSLT-prosessorikohtaisiin ratkaisuihin, mikä estää esimerkiksi työkalujen omien debuggereiden käytön.

Yleiskäyttöisiä osia voidaan molemmilla tekniikoilla toteuttaa ja ottaa käyttöön helposti. JSP-tekniikka tarjoaa tähän erityisen paljon erilaisia ratkaisutapoja, joista kannattaa kylläkin valita käytettäväksi toimivimmat. Omien tagikirjastojen toteuttaminen olisi tässä ainakin sivujen selkeyden ja tehtävien jakamisen kannalta paras ratkaisu, joskin myös suoralla sisällyttämisellä tai kutsumisella on etunsa joissakin tilanteissa.

Lokalisointi onnistui elegantisti molemmilla tekniikoilla. XSLT:llä valittiin tehokas mutta jonkin verran ylimääräistä koodia tuottava toteutustapa. JSP:llä toteutus tehtiin suoraviivaisesti JSTL:n erityisesti tähän tarjoaman mekanismin avulla. Molemmat ratkaisut ovat suoritusaikaisia.

Debuggaus on ongelmallista molempien tekniikoiden kanssa. XSLT-debuggereita on olemassa, mutta ne rajaavat XSLT-prosessorikohtaiset laajennukset pois. JSP-

tagikirjastojen debuggaus on ylipäätään vaikeaa. Jos tagikirjaston lähdekoodin on osana kehitettävää projektia, voidaan debuggaus kohdistaa myös kirjaston toteuttaviin luokkiin. Molemmissa ratkaisuissa joudutaan helposti tilanteeseen, jossa käytännössä helpoin tapa selvittää ongelmia ovat koodiin kirjoitetut debug-tulosteet. Parempia keinoja kaivattaisiin.

JSP-toiminnallisuuden laajentaminen onnistuu kätevästi tagikirjastotekniikalla. Tähän voidaan toteuttaa kokonaan omia tagikirjastoja tai käyttää valmiita. XSLT:n laajentamisessa ajaudutaan helposti XSLT-prosessorikohtaisiin ratkaisuihin. Tämä on kierrettävissä, mutta ratkaisu ei ole järin kaunis.

XSLT-tekniikka erottaa käyttöliittymän ja liiketoimintalogiikan kehittämisen paremmin kuin JSP. Tämän johtuu yksinkertaisesti siitä, että normaalin ohjelmalogiikan ohjelmointi on XSLT:llä vaikeaa. JSP-sivuille sen sijaan voidaan kirjoittaa mitä tahansa Java-ohjelmakoodia, mikä houkuttelee rikkomaan MVC-arkkitehtuurin mukaisen komponenttien vastuunjaon.

XML-käteismuistin toteuttaminen onnistui molemmilla tekniikoilla. XSLTC:n kanssa voidaan käyttää sen sisäistä esitystapaa lähdedokumenteille ja saada tästä tehokkuusetua. JSP-ratkaisussa joudutaan käyttämään Apachen JSTL-kirjaston hyödyntämää W3C DOM-mallia ja toteuttamaan omat synkronointimekanismit.

Molempia ratkaisuja toteuttaessa tuli selväksi, ettei kehittäjälle riitä osata pelkästään juuri valittua tekniikkaa. On tunnettava myös laajennustapoja, jotka usein johtavat muiden tekniikoiden, tässä tapauksessa Java-luokkien käyttämiseen. Lisäksi XSLT-muunnosprosessoreiden ja erityisesti valitun prosessorin toimintaa on ymmärrettävä ja ongelmia osattava selvittää. Myös JSTL-toteutuksen kanssa tulee vastaan tilanteita, joissa ei riitä käyttöliittymäosaaminen vaan tarvitaan ohjelmointitaitoja.

Valituissa tuotteissa esiintyi ongelmia, jotka vaikeuttivat toteutustyötä. Xalan on toimivuudeltaan yllättävän heikossa kunnossa, ottaen huomioon miten pitkään sitä on jo kehitetty. Myös XSLTC-kääntäjän kanssa törmättiin vähäisempiin ongelmiin ja toteutettu käteismuistiratkaisu ei toiminutkaan enää XSLTC:n seuraavalla versiolla. JSTL-toteutukseen kuuluva Jaxen-XPath-prosessori ei tukenut vielä kaikkia XPath-kielen ominaisuuksia, eikä nimiavaruuksia saatu käyttöön toivotulla tavalla.

8.1.2 Yhteenveto

Yhteenvetona voidaan sanoa, että ratkaisujen toteutus onnistui molemmilla tekniikoilla kohtuullisella vaivannäöllä. Harmillisena ongelmana ovat toteutusten ohjelmavirheet ja puutteellisuudet, joita joudutaan kiertämään.

XSLTC-kääntäjä XSLT-toteutuksena on riittävän kypsä käyttöönotettavaksi yksinkertaisen ratkaisun toteuttamiseen. Yksinkertaisella tässä tarkoitetaan ratkaisua, jossa ei liiemmin käytetä laajennusmahdollisuuksia, jotta esimerkiksi debuggaus voidaan tehdä muilla välineillä. Toimivuudeltaan XSLTC vaikutti näissä testeissä paremmalta kuin mitä yleisesti voidaan julkaistuista testituloksista päätellä. Eri versioissa on omat ongelmansa.

JSTL ei sellaisenaan aivan riitä ratkaisuksi kovin monimutkaiseen XML-käsittelyyn tai käsittely on käytännössä kätevämpää tehdä laajentamalla kirjastoa omilla tageilla. Näin tehtäessä JSTL:n käyttämä DOM-malli asettaa rajat XML-käsittelylle. Tämä mahdollistaa monimutkaisen käsittelyn mutta on hankalahko ohjelmoida. Toinen vaihtoehto on toteuttaa itse tai käyttää jotain valmista, ilmaisuvoimaisempaa tai tilanteen mukaan räätälöityä XML-tagikirjastoa. JSTL:n ydin- ja muotoilu- ja lokalisointikirjastoja voidaan toki käyttää apuna muuten.

8.2 Suorituskykymittausten tausta

8.2.1 Tavoitteet

Näillä mittauksilla pyrittiin vertailemaan käännettyjen XSLT-dokumenttien ja JSP-sivujen suorituskykyä. Haettiin vastausta kysymykseen, olisivatko tekniikat tehokkuuden puolesta kypsiä tuotantokäyttöön. Kaikki testit ajettiin molemmilla tekniikoilla toteutetuille ratkaisuille. Lisäksi pyrittiin mittaamaan XML-dokumenttien validoinnin tuomaa lisäkuormaa.

Kuriositeettina kokeiltiin myös HotSpot-virtuaalikoneen sisältämää vaihtoehtoista, inkrementaalia roskankeruualgoritmia. Sen suorituskyvyn testaaminen sadan prosentin prosessorikuormalla ei kuitenkaan ole paras mahdollinen ratkaisu, sillä tämä algoritmi haukkaa varsin suuren osan prosessoriajasta, lisäten näin kokonaissuoritusaikaa merkittävästi.

8.2.2 Vakioidut parametrit

Testit ajettiin Sun HotSpot-virtuaalikoneen palvelinversiolla (parametri –server). Virtuaalikoneen suorittamiin optimointeihin ei puututtu, vaan käytettiin tässä oletusarvoja. HotSpot-optimointi oli käytössä.

Tomcat-palvelimessa asetettiin säikeiden vähimmäismääräksi 5 ja maksimimääräksi 25. Testattava sovellus oli ainoa palvelimelle asennettu. Lokikirjoitukset pyrittiin minimoimaan. Tagikirjastojen tagien ilmentymien käteismuisti (pooling) asetettiin päälle. Sovellusten ajonaikaiseen, automaattiseen päivittämiseen liittyvät ominaisuudet kytkettiin pois käytöstä, eli esimerkiksi JSP-sivujen aikaleimoja ei tarkistettu eikä JSP-sivuja voinut päivittää testien suorituksen aikana.

XSLTC-kääntäjä parametroitiin siten, että se käänsi käytettävät XSLT-dokumentit ensimmäisellä kutsulla. Tämän jälkeen ei tiedostoja ladattu levyltä.

8.2.3 Testisovellus ja -laitteisto

Tehokkuusmittauksia varten toteutettiin testisovellus. Se paketoitiin J2EE-standardin mukaiseen WAR-pakettiin. Sovelluksessa oli mukana kaikki toteutetut luokat molempien ratkaisujen testaamiseen sekä joukko JSP-sivuja. XML, XSLT ja Schema-tiedostoja ei paketoitu mukaan sovellukseen vaan ne ladattiin työhakemistosta.

Testien ajamista varten sovellukseen toteutettiin erityinen JSP-sivu, jonka avulla pystyttiin parametroimaan mm. ajettavien iteraatioiden lukumäärä ja testeissä

käytettävät XML, XSLT ja JSP-tiedostot. Kyseinen JSP tuotti tulokset selaimelle CSV-muotoon (comma-separated values). XML Schema-tekniikkaan perustuvan validoinnin suorituskyvyn testaamiseksi toteutettiin yksinkertainen skeemadokumentti.

Testilaitteiston prosessorina oli AMD Athlon XP 1700+ ja laitteistossa oli 512 megatavua PC2100-muistia. Käyttöjärjestelmä oli Windows XP Professional.

8.2.4 XML-dokumentit

Tiedostoja ei voida tässä esittää, sillä niissä esiintyy runsaasti vielä julkaisemattoman verkkopalvelun mukaista materiaalia. Testiaineistona käytettiin todellista sisältöä, jotta ratkaisujen toteutettavuudesta ja tehokkuudesta saataisiin tietoa nimenomaan kohdeyrityksen tarpeita ajatellen.

Testiaineistoon kuuluu kolme XML-dokumenttia; varsinainen lähdedokumentti, virheilmoituksia sisältävä dokumentti ja viitteellä haettavan taulukon sisältävä dokumentti. Lähdedokumentissa on 119 elementtiä ja sen koko on noin 5,5 kilotavua. Suurin osa elementeistä on erilaisissa hierarkioissa ja tasoja on kaikkiaan viisi. Ilmoitusdokumentissa on kaksi suomenkielistä ja kaksi ruotsinkielistä virheilmoitusta. Viitedokumentissa on kolme suomenkielistä ja kolme ruotsinkielistä viitattua elementtiä.

8.2.5 Dokumenttien käsittely

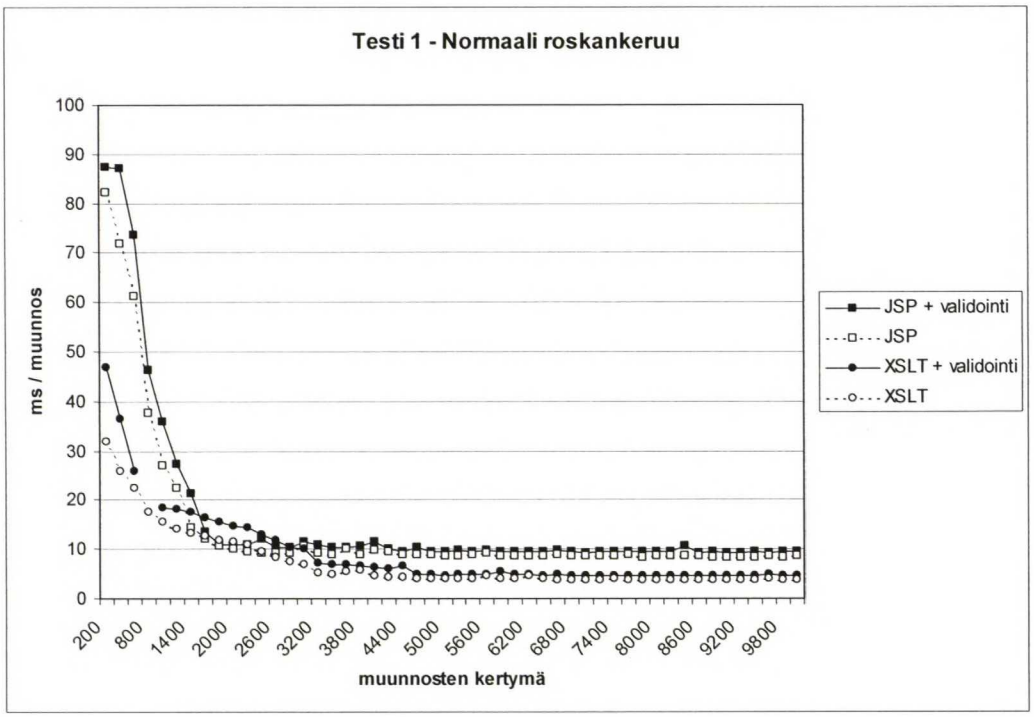
Suurin osa lähdedokumentin aineistosta tulostetaan joko käsittelemättömänä tai muotoiltuna osaksi XSLT ja JSP-tekniikoilla tuotettavaa käyttöliittymäsivua. Käsittely tapahtuu luvussa seitsemän kuvatuilla mekanismeilla. Tietoja muotoillaan, taulukoita iteroidaan ja virheilmoitukset ja viitteet haetaan erillisistä dokumenteista. Kaikki lokalisoitavat merkkijonot (24 tietoa) haetaan lokalisointitiedostoista. Lopputuloksena syntyy XHTML-muotoinen käyttöliittymäsivu, jolle on koottu joukko erilaisia kohdeyrityksen verkkopalvelun käyttöliittymissä esiintyviä elementtejä.

8.3 Suorituskykymittaukset

Käytetyt työkalut esitellään luvun seitsemän alussa, osana ratkaisujen kuvausta.

8.3.1 Testitapaus 1 – Pitkän ajan optimoituminen

Tässä testissä ajettiin 50 kertaa 200 muunnoksen sarja. Mittaustuloksina kirjattiin kunkin 200 muunnoksen sarjan osalta yhteen muunnokseen keskimäärin kulunut aika. Kukin testiajo ajettiin viisi kertaa ja tuloksena esitetään näiden ajojen keskiarvo. Virtuaalikoneelle määriteltiin yllä kuvattujen vakioarvojen lisäksi Java-olioiden kekorakenteen kooksi 100 megatavua. Testit ajettiin JSP- ja XSLT-tekniikoilla, Schema-validoinnilla ja ilman sitä.



Kuva 8-1: Testitapaus 1

Alkupään tulosten varianssi eri iteraatioiden välillä on suuri. Tarkempi vertailu kannattaa siten perustaa 20 viimeisen sarjan keskiarvoon. Lisäksi tuloksiin vaikuttaa se, kuinka monen muunnoksen välein suoritusaikaa mitataan ja tässä mittausväli oli valittu 200:ksi.

JSP-testien tulosten keskiarvot ovat ajon alussa validoinnin kanssa noin 87 ms per muunnos ja ilman validointia 82 ms. Suorituskyky paranee kuitenkin nopeasti ja saavuttaa 10 ms tason noin 2000 muunnoksen jälkeen. Validoinnin kanssa tulos stabiloituu noin 9,5 ms:iin ja ilman validointia 8,5 ms:iin, kun nämä lasketaan 20 viimeisen tuloksen keskiarvona. Näiden tarkoista arvoista laskettuna validoinnin tuoma lisäys aikaan on noin 11 %.

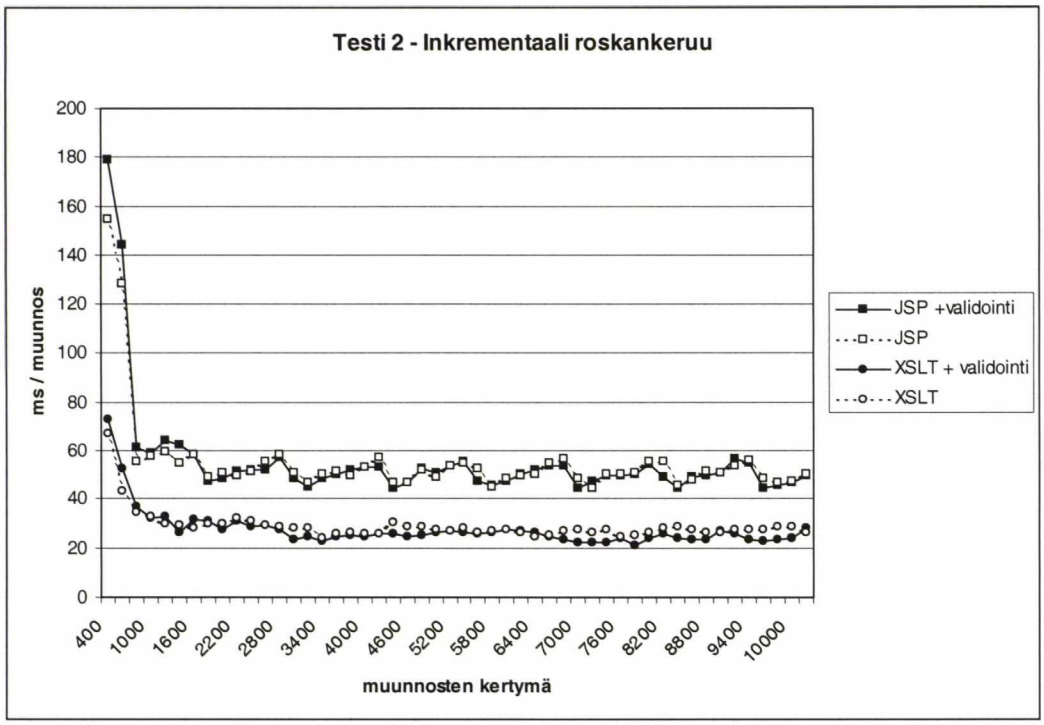
XSLT-testien tulosten keskiarvot ovat alussa validoinnilla 46 ms ja ilman validointia 32 ms. Stabiloituminen kestää pidempään kuin JSP:llä, mutta saavuttaa matalamman tason; validoinnin kanssa 20 viimeisen tuloksen keskiarvo on 4,6 ms ja ilman validointia 3,8 ms. Validointi hidastaa tässä noin 19 %.

Testien perusteella XSLT-toteutus on suorituskyvyltään noin kaksinkertainen JSP-toteutukseen nähden. Virtuaalikoneen optimoinnit vaikuttavat huikeasti, mutta vaativat tuhansia muunnoksia ennen kuin tulokset stabiloituvat lopulliselle tasolle.

8.3.2 Testitapaus 2 – Inkrementaali roskankeruu

Tässä, kuten edellisessäkin testissä, ajettiin 50 kertaa 200 muunnoksen sarja. Mittaustuloksina kirjattiin kunkin 200 muunnoksen sarjan osalta yhteen muunnokseen keskimäärin kulunut aika. Validoinnin kanssa samat testit iteroitiin kolme kertaa ja ilman validointia viisi kertaa. Tuloksena esitetään näiden keskiarvot. Virtuaalikoneelle määriteltiin yllä kuvattujen vakioarvojen lisäksi

Java-olioiden kekorakenteen kooksi 100 megatavua ja käytössä oli inkrementaali roskankeruu.



Kuva 8-2: Testitapaus 2

Tässäkin alkupään tulosten varianssi eri iteraatioiden välillä on varsin suuri. Tarkempi vertailu kannattaa siten perustaa viimeisimpien sarjojen keskiarvoihin. Samoin kuin edellisessä testitapauksessa, tässäkin valittiin mittausväliksi 200 muunnosta.

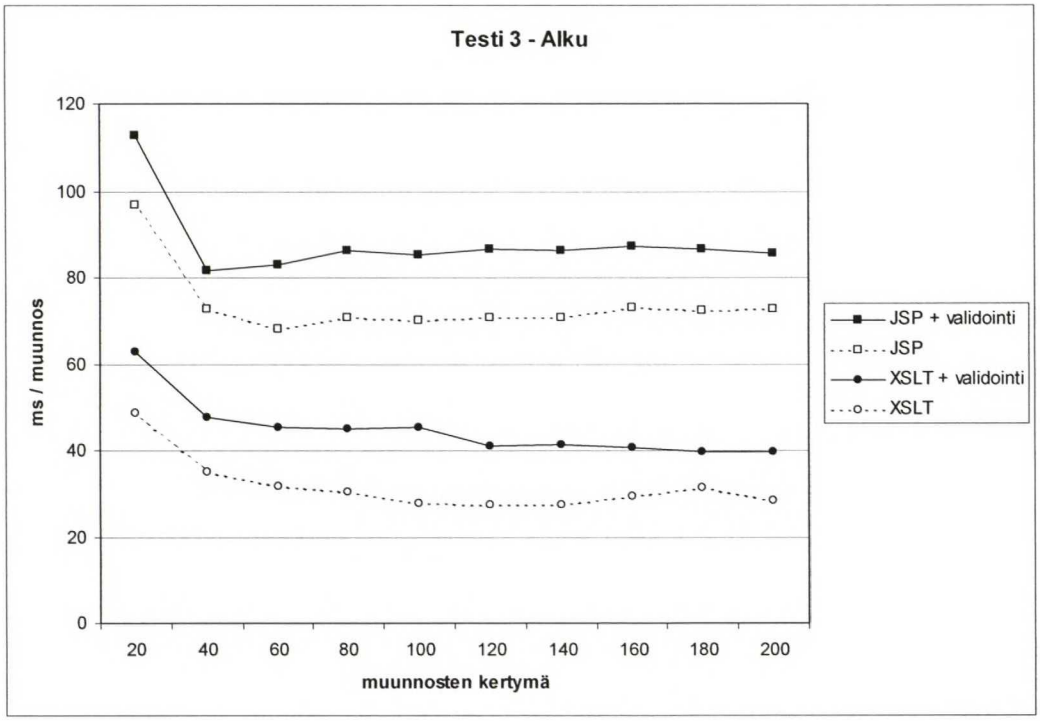
JSP-testien tulosten keskiarvot ovat alussa 179 ms validoinnin kanssa ja 154 ms ilman validointia. Tulokset eivät stabiloidu inkrementaalin roskankeruualgoritmin ollessa käytössä kovin hyvin, mutta viimeisten 20 tuloksen osalta keskiarvo on validoinnin kanssa 50 ms ja ilman validointia 51 ms.

XSLT-testien tulosten keskiarvot ovat alussa 73 ms validoinnilla ja 67 ms ilman validointia. Tulokset stabiloituvat paremmin kuin JSP-toteutuksessa ja 20 viimeisen tuloksen osalta keskiarvo validoinnilla on 24 ms ja ilman validointia 27 ms.

Testien perusteella suorituskyky saavuttaa inkrementaalin roskankeruun ollessa käytössä lopullisen tasonsa nopeammin kuin oletusarvoisella roskankeruulla. Tulokset eivät kuitenkaan stabiloidu yhtä hyvin ja 20 viimeisen tuloksen osalta aikaa kuluu sekä JSP- että XSLT-ratkaisussa yli viisinkertaisesti normaaliin roskankeruuseen verrattuna. Lisäksi huomattavaa ja jokseenkin selittämätöntä tässä tuloksessa on, ettei validointi hidastanut muunnoksia käytännössä lainkaan.

8.3.3 Testitapaus 3 – Suorituskyky alussa

Tässä testissä ajettiin 10 kertaa 20 muunnoksen sarja. Mittaustuloksina on kunkin sarjan osalta keskimääräinen yhteen muunnokseen kulunut aika. Testit ajettiin viisi kertaa ja tuloksena esitetään ajojen keskiarvo. Virtuaalikoneelle asetettiin käytössä oleva muistin määrä riittävän suureksi (250 megatavua), jotta roskankeruuta ei tapahtuisi testien suorituksen aikana.



Kuva 8-3: Testitapaus 3

JSP-testien tulos validoinnilla keskiarvona kaikista tässä ajetuista testeistä on noin 88 ms ja ilman validointia 74 ms. XSLT:llä vastaavat luvut ovat 45 ms ja 32 ms. Tämä tulos tuo esiin muunnosten hitauden heti palvelimen käynnistyksen jälkeen.

8.4 Suorituskykymittausten analyysi

8.4.1 Yleisanalyysi

Testitapaukset 1 ja 2 tuovat hyvin esiin HotSpot-virtuaalikoneen suorittaman optimoinnin. Vaikutukset näkyvät tulosten paranemisessa huimasti ensimmäisten muutaman tuhannen muunnoksen aikana.

Kaikissa testituloksissa näkyy JSP-toteutuksen hitaus verrattuna XSLTC:llä käännettyihin XSLT-muunnoksiin. Suorituskykyero on sen verran suuri, että sitä analysoidaan tarkemmin.

JSP:n heikkouksina ovat ainakin JSTL-tagien sisältämien Expression Language ja XPath –kielisten lausekkeiden jättäminen ajonaikaisesti käsiteltäviksi. JSP-sivujen käännösprosessi ei käsittele näitä merkkijonoja. XSLTC-kääntäjä sitä vastoin käsittelee XPath-lausekkeet käännösaikaisesti ja tuottaa niistä tavukoodia siltä

osin kuin se on mahdollista. XSLTC:lle etua tuo myös sen oma sisäinen esitystapa XML-dokumenteille.

JSP-toteutus sisältää myös selvästi enemmän lähdekoodia ja haaskaa muistia suoritusaikana. Testitapaus 3:n ajaminen ilman roskankeruuta paljastaa että sen aikana luodaan JSP-ratkaisussa n. 2 750 000 uutta oliota kun vastaava luku XSLTC:llä on vain 655 000. Ero tässä on nelinkertainen.

Validoinnin vaikutus on kokonaisuuteen varsin pieni. Näiden testien perusteella validointi voitaisiin ottaa hyvinkin tuotantokäyttöön ilman merkittäviä vaikutuksia suorituskykyyn.

Inkrementaalin roskankeruun kokonaissuoritusaikaa kasvattava vaikutus oli varsin suuri. Algoritmin mahdolliset edut eivät tässä tulleet millään tavoin esiin. Tulos ei kuitenkaan sulje pois sitä, etteikö tätä algoritmia käyttämällä voitaisi tasoittaa suorituskykyä, mikäli prosessorikuorma olisi huomattavasti tässä testissä käytettyä noin sataa prosenttia pienempi.

Testitapaus 3 otettiin mukaan, jotta voitaisiin selvittää miten ratkaisut käyttäytyvät aivan palvelimen käynnistyksen jälkeen ja ilman roskankeruun aiheuttamaa lisäkuormaa. Tulokset kertovatkin muunnosten olevan alussa karkeasti arvioiden 5-10 kertaa hitaampia kuin HotSpot-virtuaalikoneen suorittamien optimointien jälkeen.

8.4.2 Tulosten luotettavuus

Yksittäiset tulokset heittelevät paljon siitä huolimatta, että testilaitteiston muu kuorma pyrittiin minimoimaan. Alkupään tuloksissa eri iteraatioiden välillä on suuri varianssi ja huomio kannattaa siten kiinnittää testitapausten 1 ja 2 osalta erityisesti 20 viimeisimmän tuloksen keskiarvoon. Näillä laajemman tulosjoukon keskiarvoilla saatiin hyvin esiin tulosten yleinen taso niiden stabiloiduttua virtuaalikoneen optimointien myötä ja pystyttiin näin vertailemaan ratkaisutapoja toisiinsa.

Tulokset koskevat juuri tässä tapauksessa käytettyjä JSP- ja XSLT-dokumentteja, XML-jäsentimiä ja kääntäjiä. JSP-käännökset tehtiin Tomcat-palvelimeen kuuluvalla Jasperilla ja XSLT-käännökset Apache-projektin XSLTC-kääntäjällä.

Tekniikoiden erilaisuus ei vaikuttanut ainoastaan muunnokseen vaan myös XML-dokumenttien jäsentämiseen ja validointiin. Ratkaisut oli räätälöity molemmille tekniikoille sopivaksi. Erityisesti JSP-tekniikalla voitaneen toteuttaa tehokkaampiakin ratkaisuja, jos ei tukeuduta JSTL-tagikirjastoon XML-käsittelyssä. XSLTC:n kanssa jouduttiin käyttämään XML-dokumenttien jäsentämisessä suoraan SAX-rajapinnan toteuttavaa parseria, kun validointi oli kytkettynä päälle. Muuten käytettiin suoraan XSLTC:n sisäistä käsittelytapaa.

9. Luku - Yhteenveto

9.1 Työn yhteenveto

Portaaliympäristöissä joudutaan käsittelemään hyvin erilaisista järjestelmistä vastaanotettuja sanomia tai vastauksia. Näitä ja muita portaalin käytössä olevia tietoja hyväksi käyttäen tuotetaan organisaation tarjoamiin palveluihin yhtenäinen käyttöliittymä. Tähän soveltuvia tekniikoita ja erilaisia arkkitehtuurimalleja on lukemattomia. Tässä diplomityössä vertailtiin kahta erilaista XML-dokumenttien käsittelyyn perustuvaa toteutustapaa. Molemmat ratkaisut toimivat J2EE-ympäristössä ja tukeutuvat ilmaisiin, suurimmalta osin avoimen lähdekoodin komponentteihin.

Ensimmäinen sovellettava tekniikka oli XML-käsittelyyn erikoistunut XSLT. XSLT on standardi, jonka toteuttavana ratkaisuna käytettiin Apache-projektissa kehitettävää XSLT-dokumentit Java-tavukoodiksi kääntävää XSLTC-ohjelmaa. Toinen ratkaisu toteutettiin Java-ympäristössä yleisesti käyttöliittymien tuottamiseen tarkoitetulla JSP-tekniikalla, jonka toteutuksena käytettiin niin ikään Apache-projektiin kuuluvaa Tomcat-palvelinohjelmistoa. Erityisesti JSP:n yhteyteen valittiin käyttöön standardoitu JSTL-laajennuskirjasto ja pyrittiin hyödyntämään sitä mahdollisimman tehokkaasti.

Ratkaisuja vertailtiin ja arvioitiin kehittämisen helppouden ja kustannustehokkuuden sekä toisaalta tuotantokäytössä tärkeän kriteerin, suorituskyvyn kannalta. Tavoitteena oli selvittää tekniikoiden soveltuvuutta kohdeyrityksen verkkopalvelun toteutuksessa käytettäväksi. Samalla haettiin tietoa ja saatiin kokemuksia myös muista mahdollisista ratkaisuista.

Suurin osa testattaviksi valituista XML-käsittelyyn ja käyttöliittymän tuottamiseen liittyvistä käyttötilanteista saatiin toteutettua tyydyttävällä tavalla molemmilla tekniikoilla. XSLT-tekniikan ja XSLTC-kääntäjän käyttö osoittautui toimivaksi ratkaisuksi testitapausten valossa. JSP-sivuilla käytetyn JSTL-tagikirjaston XML-käsittelyominaisuudet ovat vaatimattomammat ja monimutkaisempaa XML-käsittelyä varten voidaan toiminnallisuutta laajentaa omilla tageilla. Molempiin ratkaisuihin valituissa kirjastoissa esiintyi myös virheitä ja puutteita, mitkä aiheuttivat ongelmia.

Suorituskykyä mitattiin kolmella erilaisella testitapauksella, jotka kukin ajettiin molemmilla tekniikoilla toteutetuille ratkaisuille ja erikseen myös XML Schema-

määrityksiin perustuvan validoinnin ollessa käytössä. Ensimmäisessä testitapauksessa tarkasteltiin pitkän ajan suorituskykyä hyvin rajallisessa muistiavaruudessa. Toisessa testattiin vastaavaa tilannetta HotSpot-virtuaalikoneen vaihtoehtoisen roskankeruualgoritmin ollessa käytössä. Kolmannella testitapauksella pyrittiin selvittämään suorituskykyä heti palvelimen käynnistymisen jälkeen, ennen virtuaalikoneen tekemiä optimointeja ja roskankeruualgoritmin tuomaa lisäkuormaa.

XSLTC-pohjaisen toteutuksen suorituskyky oli noin kaksinkertainen JSP/JSTL-ratkaisuun nähden. Tätä voidaan selittää ainakin JSP-tagikirjastojen käyttöön liittyvän yleisen raskauden kuten Expression Language ja XPath –lausekkeiden ajonaikaisen tulkkauksen ja toisaalta XSLTC:n tehokkaan XML-käsittelyn vaikutuksella.

HotSpot-virtuaalikoneen suorittamat optimoinnit tulivat testeissä hyvin esiin ja suorituskyky stabiloitui molemmissa ratkaisuissa muutaman tuhannen muunnoksen jälkeen 5-10 kertaa tehokkaammaksi alkutilanteeseen verrattuna. XML Schema –dokumentteihin perustuva validointi vaikutti pitkällä aikavälillä ajankäyttöön vain noin 10-20 prosenttia. Virtuaalikoneen vaihtoehtoinen roskankeruualgoritmi, joka pyrkii lomittumaan tasaisemmin suorituksen yhteyteen, hidasti testejä merkittävästi.

Alla esitetään joitakin mahdollisia jatkotutkimuksen kohteita, liittyen ratkaisujen ja erityisesti JSP-ratkaisun parantamiseen.

9.2 Jatkotutkimusmahdollisuuksia

Sekä erityisesti XML-käsittelyyn liittyviin tekniikoihin että JSP-tekniikkaan on tulossa ja tullut laajojakin muutoksia ja parannuksia. Muutokset ovat niin tuoreita, että toteutuksia on saatavilla vielä hyvin rajallisesti. Tästä syystä uusimpia tekniikoita ei otettu vielä tähän työhön vertailtaviksi. Jatkossa ratkaisujen toteutettavuutta ja suorituskykyä kannattaisi arvioida myös XSLT 2.0 ja JSP 2.0 –tekniikoilla.

JSTL-kirjaston XML-käsittelyyn liittyvät ominaisuudet osoittautuivat vaatimattomiksi. JSTL-kirjaston uutta versiota ja erityisesti XML-käsittelyyn erikoistuneita tagikirjastoja voidaan vertailla tässä työssä käytettyihin tekniikoihin. Esimerkiksi dom4j-kirjaston päällä toimiva Apache XTags on ilmaisuvoimaisempi. Myös kaupallisten J2EE-sovelluspalvelinten mukana toimitetaan XML-kirjastoja. Oman räätälöidyn XML-tagikirjaston toteuttaminen voi myös tulla kyseeseen.

XSLTC-kääntäjä perustuu avoimeen lähdekoodiin. Sillä on myös kaupallinen kilpailija Gregor, joka on joidenkin suorituskykymittausten mukaan selvästi tehokkaampi. Sen käyttöä kannattaa harkita, mikäli XSLTC:n suorituskyky tai puutteet osoittautuvat liian suuriksi ongelmiksi.

XQuery on varteenotettava vaihtoehto XSLT:lle yleisten XML-muunnosten toteutukseen. Se voi olla toimiva ratkaisu myös nimenomaan käyttöliittymän tuottamiseen.

A Viitteet

[1] Ambroziak, J.R. *Gregor/XSLT – Introduction*. Ambrosoft, Inc. 5.8.2003. [Viitattu 5.6.2004]. Saatavissa: http://www.ambrosoft.com/gregorxslt.htm

[2] Apache Software Foundation (The). *Apache Taglibs – XTags Library*. [Viitattu 5.6.2004]. Saatavissa: http://jakarta.apache.org/taglibs/doc/xtags-doc/intro.html

[3] Apache Software Foundation (The). *Jasper 2 JSP Engine How To*. 2002. [Viitattu 5.6.2004]. Saatavissa: http://jakarta.apache.org/tomcat/tomcat-4.1-doc/jasper-howto.html

[4] Apache Software Foundation (The). *Xerces 2 Java Parser*. 2004. [Viitattu 10.6.2004]. (Tämä sivu päivittyy uusien versioiden myötä. Käytetty Xerces-versio on 2.4.0). Saatavissa: http://xml.apache.org/xerces2-j/index.html

[5] Apache Software Foundation (The). *XSLTC Compiler Design.* 2004. [Viitattu 5.6.2004]. (Tämä sivu päivittyy uusien versioiden myötä. Käytetty XSLTC-versio on 2.5.2). Saatavissa: http://xml.apache.org/xalan-j/xsltc/xsltc_compiler.html

[6] Apache Software Foundation (The). *XSLTC Performance*. 2004. [Viitattu 5.6.2004]. (Tämä sivu päivittyy uusien versioiden myötä. Käytetty XSLTC-versio on 2.5.2). Saatavissa: http://xml.apache.org/xalan-j/xsltc/xsltc_performance.html

[7] Becker, O. *XSLT Stylesheets – Useful Things and Other Jokes*. Humboldt University Berlin. [Viitattu 15.3.2004]. Saatavissa: http://www.informatik.hu-berlin.de/~obecker/XSLT/

[8] Berners-Lee, T. Fielding, R. Irvine, U.C. Masinter, L. *RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax*. IETF. Elokuu 1998. [Viitattu 3.6.2004]. Saatavissa: ftp://ftp.rfc-editor.org/in-notes/rfc2396.txt

[9] Bondre, P. *XSLT – Efficient Programming Techniques*. XML.org. 17.4.2002. [Viitattu 24.2.2004]. Saatavissa: http://www.xml.org/xml/xslt_efficient_programming_techniques.pdf

[10] Bothner, P. *Generating XML and HTML using XQuery.* XML.com. 23.12.2002. [Viitattu 24.2.2004]. Saatavissa: http://www.xml.com/pub/a/2002/12/23/xquery.html

[11] Bray, T. Hollander, D. Layman, A. 1999. *Namespaces in XML*. W3C Recommendation. Saatavissa: http://www.w3.org/TR/1999/REC-xml-names-19990114/

[12] Bray, T. Paoli, J. Sperberg-McQueen, C.M. Maler, E. Yergeau, F. 2004. *Extensible Markup Language (XML) 1.0 (Third Edition),* W3C Recommendation. Saatavissa: http://www.w3.org/TR/2004/REC-xml-20040204/

[13] Brownell, D. *Simple API for XML*. Sourceforge.net. [Viitattu 3.6.2004]. Saatavissa: http://www.saxproject.org/

[14] Bruchez, E. Tazi, O. *Integrating JSP/JSF and XML/XSLT: The Best of Both Worlds.* TheServerSide.com. Helmikuu 2003. [Viitattu 24.2.2004]. Saatavissa: http://www.theserverside.com/articles/printfriendly.tss?l=BestBothWorlds

[15] Burke, E.M. *XSLT Processing with Java*. ONJava.com. [Viitattu 3.6.2004]. Saatavissa: http://www.onjava.com/lpt/a/1060

[16] Clark, J. 1999. *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation. Saatavissa: http://www.w3.org/TR/1999/REC-xslt-19991116

[17] Clark, J. DeRose, S. 1999. *XML Path Language (XPath)*. W3C Recommendation. Saatavissa: http://www.w3.org/TR/1999/REC-xpath-19991116

[18] Cowan, J. Tobin, R. 2004. *XML Information Set (Second Edition)*. W3C Recommendation. Saatavissa: http://www.w3.org/TR/2004/REC-xml-infoset-20040204/

[19] Delisle, P. *JavaServer Pages™ Standard Tag Library*. Sun Microsystems. Kesäkuu 2002. [Viitattu 3.6.2004]. Saatavissa: http://jcp.org/aboutJava/communityprocess/final/jsr052/index.htmlJSR-000052

[20] Dunning, John. *Developing Enterprise XSL Stylesheets – Best Practices and Lessons Learned.* XML 2002 Proceedings by deepX. 6.4.2001. [Viitattu 24.3.2003]. Saatavissa: http://www.idealliance.org/papers/xml02/dx_xml02/papers/06-04-01/06-04-01.pdf

[21] Evans, C.C. *XSLT's Template Dispatch*. The Cover Pages. 1.12.2000. [Viitattu 5.6.2004]. Saatavissa: http://www.oasis-open.org/cover/evansTemplateDispatch.html

[22] Fitzgerald, M. St.Laurent, S. (toimittaja). 2004. *Learning XSLT.* 2. painos. Sebastopol, USA. O'Reilly & Associates, Inc. 352s. ISBN 0-596-00327-7.

[23] Hanna, P. 2003. *JSP 2.0: The Complete Reference*. Berkeley, CA, USA. The McGraw-Hill Companies. ISBN 0-07-222437-1

[24] He, H. *What is Service-Oriented Architecture?* O'Reilly webservices.xml.com. 2003. (Online). 30.9.2003. [Viitattu 7.3.2004]. Saatavissa: http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html

[25] Johnson, E. 2001. *XML Usage Patterns.* XMLEurope 2001. Berliini, Saksa, 21-25.5.2001. Graphic Communications Association. [Viitattu 20.3.2004]. Saatavissa: http://www.gca.org/papers/xmleurope2001/papers/pdf/sid-01-5.pdf

[26] Kay, M. H. 2003. *XML Five Years On: A Review of the Achievements So Far and the Challenges Ahead.* Teoksessa: Proceedings of the 2003 ACM Symposium on Document Engineering. Grenoble, Ranska 20-22.11.2003. New York, USA. ACM Press. s. 29-31. ISBN 1-58113-724-9

[27] Kepser, S. *A Proof of the Turing-completeness of XSLT and XQuery.* Tübingen, Saksa. Tübingenin yliopisto. 13.5.2002. [Viitattu 24.2.2004]. Saatavissa: http://tcl.sfs.uni-tuebingen.de/~kepser/papers/xsltxq.pdf

[28] Kuchhal, R. *J2EE application performance optimization.* JavaWorld. 17.5.2004. [Viitattu 5.6.2004]. Saatavissa: http://www.javaworld.com/javaworld/jw-05-2004/jw-0517-optimization_p.html

[29] Le Hégaret, P. *Document Object Model (DOM).* W3C. 7.4.2004. [Viitattu 3.6.2004]. Saatavissa: http://www.w3c.org/DOM/

[30] Mahmoud, Q.H. *Developing Web Applications With JavaServer Pages 2.0.* Sun Microsystems. Kesäkuu 2003. [Viitattu 5.6.2004]. Saatavissa: http://java.sun.com/developer/technicalArticles/javaserverpages/JSP20/

[31] Mangano, S. St.Laurent, S. (toimittaja). 2003. *XSLT Cookbook.* Sebastopol, USA. O'Reilly & Associates, Inc. 654s. ISBN 0-596-00372-2

[32] Marx, D. *More JSP Best Practices.* JavaWorld. 25.7.2003. [Viitattu 8.6.2004]. Saatavissa: http://www.javaworld.com/javaworld/jw-07-2003/jw-0725-morejsp_p.html

[33] McGovern, James. Bothner, Per. Cagle, Kurt. Linn, James. Nagarajan, Vaidyanathan. 2004. *XQuery Kick Start.* USA. Sams Publishing. ISBN 0-672-32479-2

[34] Metastuff Ltd. *dom4j.* 8.6.2004. [Viitattu 18.6.2004]. (Tämä sivu päivittyy uusien versioiden myötä). Saatavissa: http://dom4j.org/

[35] Nash, M. 2003. *Java Frameworks and Components – Accelerate your Web Application Development.* Cambridge, Iso-Britannia. Cambridge University Press. ISBN 0-521-52059-2

[36] Novachev, D. *The Functional Programming Language XSLT - A proof through examples.* TopXML. Marraskuu 2001. [Viitattu 2.5.2004]. Saatavilla: http://www.topxml.com/xsl/articles/fp/fp.zip

[37] Pawson, D. *XSLT Questions and Answers.* 1999-2004. [Viitattu 5.6.2004]. Saatavissa: http://www.dpawson.co.uk/xsl/sect2/sect21.html

[38] Pelegrí-Llopart, E. *JavaServer Pages™ Specification: Version 1.2.* Sun Microsystems. 27.8.2001. [Viitattu 3.6.2004]. Saatavissa: http://www.jcp.org/aboutJava/communityprocess/final/jsr053/

[39] Punte, S. *Fast XSLT.* XML.com. 2.4.2003. [Viitattu 3.6.2004]. Saatavissa: http://www.xml.com/lpt/a/2003/04/02/xsltc.html

[40] Quin, L. 2004. *The Extensible Stylesheet Language Family (XSL).* W3C. 11.5.2004. [Viitattu 3.6.2004] Saatavissa: http://www.w3.org/Style/XSL/

[41] Ray, E.T. 2003. St.Laurent, S. (toimittaja). *Learning XML – Creating Self-Describing Data.* Sebastopol, USA. O'Reilly & Associates, Inc. 400s. ISBN 0-596-00420-6

[42] Schott, S. Noga, M.L. 2003. *Lazy XSL Transformations.* Teoksessa: Proceedings of the 2003 ACM Symposium on Document Engineering. Grenoble, Ranska 20-22.11.2003. New York, USA. ACM Press. s. 9-18. ISBN 1-58113-724-9. Saatavissa myös: http://www.info.uni-karlsruhe.de/papers/schott-noga-lazy-xslt-2003.pdf

[43] Seshadri, G. *Understanding JavaServer Pages Model 2 Architecture.* Joulukuu 1999. [Viitattu 5.6.2004]. Saatavissa: http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc_p.html

[44] Shannon, B. *J2EE 1.3 Specification.* Sun Microsystems. 27.7.2001. [Viitattu 3.6.2004]. Saatavissa: http://java.sun.com/j2ee/j2ee-1_3-fr-spec.pdf

[45] Shannon, B. *J2EE 1.4 Specification.* Sun Microsystems. 24.11.2003. [Viitattu 3.6.2004]. Saatavilla: http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf

[46] Spielman, S. 2004. *JSTL - Practical Guide for JSP Programmers.* San Francisco, CA, USA. Morgan Kaufmann. ISBN 0-12-656755-7

[47] Stewart, C. Bayes, C. Fuller, J. Ogbuji, U. Pawson, D. Tennison, J. *EXSLT.* [Viitattu 3.6.2004]. Saatavissa: http://www.exslt.org/

[48] Sun Microsystems. *J2EE Platform Technologies.* J2EE BluePrints. 2001. [Viitattu 12.6.2004]. Saatavissa: http://java.sun.com/blueprints/guidelines/designing_enterprise_applications/platform_technologies/index.html

[49] Sun Microsystems. *Java API for XML Processing (JAXP).* 2004. [Viitattu 5.6.2004]. Saatavissa: http://java.sun.com/xml/jaxp/index.jsp

[50] Sun Microsystems. *JavaServer Faces Technology – Documentation.* 2004. [Viitattu 5.6.2004]. Saatavissa: http://java.sun.com/j2ee/javaserverfaces/reference/docs/index.html

[51] Sun Microsystems. *JSR-045: Debugging Support for Other Languages.* 24.11.2003. [Viitattu 3.6.2004]. Saatavissa: http://jcp.org/en/jsr/detail?id=45

[52] Sun Microsystems. *The Java HotSpot Virtual Machine, v1.4.1.* Syyskuu 2002. [Viitattu 3.6.2004]. Saatavilla: http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/JHS_141_WP_d2a.pdf

[53] W3C. *XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition).* World Wide Web Consortium. 1.8.2002. [Viitattu 12.3.2004] http://www.w3.org/TR/2002/REC-xhtml1-20020801/

[54] Williams, K. *XML for Data: XSL style sheets: push or pull?* IBM developerWorks. 1.5.2002. [Viitattu 5.6.2004]. Saatavissa: http://www-106.ibm.com/developerworks/xml/library/x-xdpshpul.html?open&l=976

[55] Zdun, U. 2002. *Dynamically Generating Web Application Fragments from Page Templates.* Teoksessa: Proceedings of Symposium of Applied Computing (SAC 2002). Madrid, Espanja, 11-14.3.2002. New York, USA,. ACM Press. s.1113-1120. ISBN 1-58113-445-2. Saatavissa myös: http://wi.wu-wien.ac.at/~uzdun/publications/pageTemplates.pdf

B Liitteet

B.1 XSLT-muunnoskomponentti

Konfigurointi:

```
# true, false = käytetäänkö XML Schemoihin perustuvaa validointia
schema_validointi=false

# UTF-8, ISO-8859-1, vaikuttaa vain jos käytössä on
# OutputStream, eikä Writer. Writerin koodaus määritellään
# oliota luotaessa
tulostus_koodaus=UTF-8

# yes, no = sisennetäänkö XML-tulostus vai meneekö putkeen
# ilman whitespaceja
tulostus_sisennys=no

# 0=ei pollata, 1-N on sekunteja levypollausten välillä.
# Jos pollataan ja class on tuhottu ja XSL on muuttunut, se käännetään
# uudelleen.
cache_levypollaus=20

# true, false = ladataanko vain luokkapolusta vai etsitäänkö
# myös xsl/*.xsl ja xsl_class/*.class tiedostoja
cache_vain_classpath=false

# true xsltc (käännös), false xalan (tulkkaus)
xsltc=true

# true = XSLTCSource DOM-cachessa (Xalan 2.5.2)
# false = DOMSource DOM-cachessa (Xalan 2.6.0) 2.6.0:ssa ei jostain
# syystä enää tueta XSLTCSourcen käyttöä liitedokumenteille, joten
# käytetään DOMSourcea, joka on hitaampi
xsltc_xsltcsource=true
```

Kutsurajapinta:

```
/**
 * Tekee XSLT-muunnoksen XSLTC:llä käännetyllä muunnosluokalla.
 *
 * @param santun       XML-vastauksen tunniste
 * @param san_versio   XML-vastauksen versio
 * @param xsltNimi     XSLT-tiedoston nimi
 * @param out          Writer, johon tulos kirjoitetaan
 * @param xmlData      käsiteltävä vastaus InputStreamina
 * @param parametrit   XSLT-muunnokselle menevät parametrit
 * @param konteksti    käyttäjän konteksti
 * @throws XSLTException    kun mikä tahansa poikkeus otetaan
 *                          kiinni
```

```
 */
static public void muunna(
    String santun, String san_versio, String xsltNimi,
    Writer out, InputStream xmlData,
    Map parametrit, Map konteksti)
      throws XSLTException;
```

B.2 Päivämäärän muotoilu

XSLT-kutsu:

```
<xsl:apply-templates mode="muotoilut_paivamaara" select="pvm_syotto"/>
```

XSLT-toteutus:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Muotoilutemplate päivämäärille.
  Templatet ottavat huomioon xsi:nil -attribuutin ja mikäli sen arvo on
  true, ei tietoa formatoida vaan käytetään tietoa kuvaavan elementin
  arvo -attribuutissa olevaa tietoa sellaisenaan. -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:date="http://exslt.org/dates-and-times"
  xmlns:xsldoc="http://www.bacman.net/XSLdoc"
  exclude-result-prefixes="xsldoc"
  extension-element-prefixes="date">

<xsldoc:author>Pete Hakkarainen</xsldoc:author>

<!-- Formatoi päivämäärän haluttuun muotoon. <br>
  Kontekstisolmuna pitää olla päivämäärä muodossa YYYY-MM-DD
    <br>
  Jos EXSLT-laajennusten date:format-date() -funktiota ei ole
  käytettävissä, ja annettu formaatti on jotain muuta kuin DD.MM.YYYY,
  keskeytyy muunnos!
-->
<xsl:template name="muotoilut_paivamaara" match="*"
mode="muotoilut_paivamaara">
<!--  <xsl:param name="paivamaara" /> -->
  <!-- VALINNAINEN päivämäärän formaatti, Javan DateFormat-luokan
       mukaisesti ilmaistuna. Oletuksena dd.MM.yyyy -->
  <xsl:param name="formaatti" select="'dd.MM.yyyy'" />
           <!-- poistaa turhat välit -->
           <xsl:text></xsl:text>
            <xsl:choose>
            <xsl:when test="./@xsi:nil = true()">
              <xsl:value-of select="./@arvo" />
            </xsl:when>
            <xsl:otherwise>
              <xsl:choose>
                <xsl:when test="function-available('date:format-date')">
                  <xsl:value-of select="date:format-date(concat(.,
'T00:00:00'), $formaatti)" />
                </xsl:when>
                <xsl:otherwise>
                  <xsl:choose>
                    <xsl:when test="$formaatti = 'dd.MM.yyyy'">
                      <xsl:value-of select="concat(substring(., 9, 2),
'.', substring(., 6, 2), '.', substring(., 1, 4))" />
                    </xsl:when>
                    <xsl:otherwise>
                      <xsl:message terminate="yes">EXSLT-laajennus
date:format-date() ei ollut käytettävissä formatointiin!</xsl:message>
```

```
                    </xsl:otherwise>
                  </xsl:choose>
                </xsl:otherwise>
              </xsl:choose>
            </xsl:otherwise>
          </xsl:choose>
</xsl:template>

</xsl:stylesheet>
```

JSTL-kutsu:

```
<x:set var="formatoitava" scope="page"
       select="$requestScope:xmlsanoma/vastaus/pvm" />
<x:set var="formatoitava_string" scope="page"
       select="string($pageScope:formatoitava)" />
<x:choose>
  <x:when select="$pageScope:formatoitava/@xsi:nil = true()">
    <x:out select="$pageScope:formatoitava/@arvo" />
  </x:when>
  <x:otherwise>
    <fmt:parseDate
      pattern="yyyy-MM-dd"
      value="${pageScope.formatoitava_string}" var="parsed" />
    <fmt:formatDate value="${parsed}" pattern="dd.MM.yyyy" />
  </x:otherwise>
</x:choose>
```

Tässä toteutuksessa on huomattavaa, että muotoilun tekeminen ei onnistu yksinkertaisella kutsulla kuten XSLT-toteutuksessa vaan muotoilun tekevään kohtaan JSP-sivulla syntyy laajasti koodia.

Lisäksi tarvitaan kaksi ilmentymää muotoiltavasta merkkijonosta, sillä JSTL:n XML-tageilla luotua muuttujaa ei voida käsitellä JSTL:n muotoilukirjaston (fmt) tageilla.

Tämän monimutkaisuuden voisi kiertää toteuttamalla kokonaan oman XML-tagikirjaston, jonka tagien kutsut olisi räätälöity täsmälleen tarpeita vastaaviksi. Ratkaisuissa kuitenkin pyrittiin hyödyntämään JSTL:ää mahdollisimman paljon.